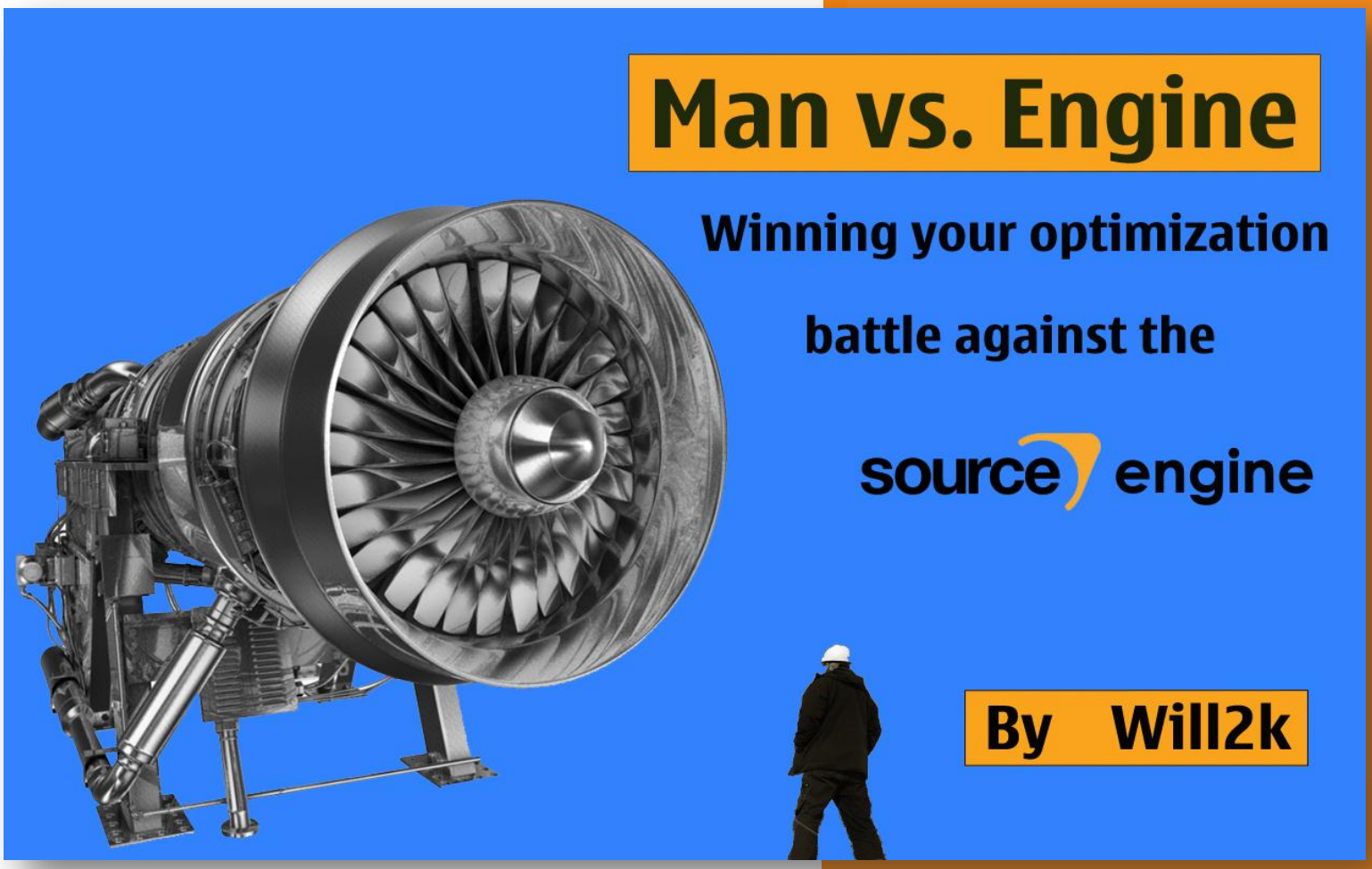

Man vs. Engine
Winning your optimization
battle against the
source engine
By Will2k

March 8, 2013

Table of Contents

I-Introduction

Process optimization is the discipline of adjusting a process so as to optimize some specified set of parameters without violating some constraints. The most common goals are minimizing cost, maximizing throughput, and/or efficiency. When optimizing a process, the goal is to maximize one or more of the process specifications, while keeping all others within their constraints.

If you have been playing my maps or reading my papers/tutorials, you would have deduced by now that I am a big fan of optimization, and it is quite honestly my favorite topic when it comes to the Source engine. But even if you haven't noticed my "intimate" relation with optimization, you will do now after reading this tutorial ☺.

You might be wondering why I am so much into optimization; the reason is very simple: Most of my career so far revolved around optimization, more specifically process optimization. How about that; not only I do optimization in the Source engine but I also do it in real life.

I started my career some 10 years ago in a multinational company, more precisely in the manufacturing plant as a process and industrial engineer doing all sorts of optimization projects on different machines to increase output/efficiency, reduce downtime, minimize loss/ scrap and improve quality. Later on I moved to another company where the scope of optimization was wider. I was now in charge of operations management where the optimization was more oriented towards full processes such as manufacturing, supply chain, quality systems and so on. Even my Master's degree thesis what about implementing a process optimization technique called "Just-in-Time" in small to medium enterprises (SME).

Optimization became second nature to me and was applicable to every aspect of what I do; that's why I feel at home when it comes to the Source engine optimization.

Make no mistake that, while visuals and gameplay are very important to any map, optimization is the deal-breaker. Players will accept a "not-so-visually-stunning" map and they will tolerate a slightly awkward layout, but no one will play a map that brings their gameplay to a slideshow of horribly-low frame rate. Making an un-optimized map means that server admins will only give it a first try and they will never touch it again.

What a waste then, of long and stressful hours spent building your level, to have no one willing to play it except maybe yourself. But fear not as I shall go through the optimization process in a methodical way, backed with screenshots (in-game and in-editor), in the hope of making you intimate with this "feared-by-beginners" process. This tutorial will put you on the right track to master the optimization process but will require perseverance from your side and lots of test maps for trial and error experience.

II-The Source Engine

If you know your enemies and know yourself, you can win a hundred battles without a single loss.
If you only know yourself, but not your opponent, you may win or may lose.
If you know neither yourself nor your enemy, you will always endanger yourself.

This, now popular idiom used by military and non-military strategists, is taken from the last verse of Chapter 3 from the book "The Art of War" written by the Chinese military general, strategist and tactician Sun Tzu (544–496 BC). The book is highly recommended to read and can prove very useful even on your personal life level.

You might be thinking that this is a nice idiom but what does it have to do with our optimization issue. Remember the title of this paper? It's winning your optimization "battle" against the Source engine; the strategy devised by Sun Tzu perfectly fits our scenario.

In order to win, you need to know yourself (skills and optimization techniques), and equally important is to know your "enemy" (the Source engine; and Valve will have to excuse me for this comparison ☺)

Despite getting many updates and facelifts lastly seen in the CSGO version, the Source engine still partly relies on rather old principles dating from the time of the original Quake engine created by John Carmack in the mid-nineties. Believe it or not, Source still has bits and pieces reminiscent of Quake.

The Source engine uses Binary Space Partition (BSP) as a method to compute visibility. BSP is a method for recursively subdividing a space into convex sets by hyperplanes. This subdivision gives rise to a representation of objects within the space by means of a tree data structure known as a BSP tree.

Binary space partitioning was developed in the context of 3D computer graphics in the 1970s, where the structure of a BSP tree allows spatial information about the objects in a scene that is useful in rendering, such as their ordering from front-to-back with respect to a viewer at a given location, to be accessed rapidly.

While some games use "visibility from a point" approach, which is mostly implemented in real time, and visibility is determined from the current viewpoint only, the BSP uses "visibility from a region" approach.

This is a kind of visibility calculation that is mostly used for indoor settings, making use of the limited viewing possibilities between rooms. The rooms are termed cells, and doors between these cells are called portals. In a pre-computation step, the data is analyzed for the potentially visible set (PVS) from all positions in a single cell.

This is done for all cells in the data set. Current implementations in games use mostly pre-computed level data in the final product, as pre-computations are too time-consuming to be done after the loading of a level. One of the benefits of this method is that it is very efficient, and that other things, like radiosity and shadows, can often be calculated in the same procedure, once the process is started out of the level editor. The main drawback is the enormous memory consumption.

The rendering process in the BSP can be described as follows: The map is carved up into convex regions that become the leaves of the BSP tree, and anytime the camera is positioned within a level, it is contained in exactly one of these convex regions. The leaves are grouped together with neighboring leaves to form clusters; exactly how these clusters are formed is determined by the tool that creates the BSP file. For each cluster, a list of all of the other clusters which are potentially visible is stored, and is referred to as the potentially visible set (PVS).

To render the map, first the BSP tree is traversed to determine which leaf the camera is located in. Once we know which leaf the camera is in, we know which cluster it is in (remembering that each leaf is contained in exactly one cluster). The PVS for the cluster is then decompressed giving a list of all the potentially visible clusters from the camera location. Leaves store a bounding box which is used to quickly cull leaves that are not within the viewing frustum.

Now if all this necessary explanation made you dizzy, then fear not as I shall display couple of screenshots where you can actually see the above in action in-game (PVS, leaves)

In this first screenshot (map de_cortona), you can see some red lines that connect with each other to form a 3D cube (in this case a rectangular cuboid to be more accurate). This 3D cube is none other than the famous leaf that we have been talking about; it's called visleaf in the Source engine and I shall be using this nomination from now on in the rest of this paper.

In this second screenshot, the 3D cube is still there but now we see other red cubes of different sizes (and shapes) that are adjacent and connected to our initial visleaf. This is the potentially visible set or PVS. It simply shows all other visleaves that the visleaf we are standing in, can "see" and have a direct line of sight as stored in the visibility matrix of the BSP tree explained earlier.

In simpler words, the visibility from a region is pre-computed during compile time and depends on the visleaf that the camera is in (the player is actually a camera) and not on the actual location where the player is standing. If you are standing in visleaf A and the pre-computed data in the PVS instructs the engine that visleaves B,C and D are visible, then the engine will draw these leaves and their content regardless of where you stand in the visleaf A, and even if you, the player, cannot directly see these visleaves.

Now you need to clear your mind and focus with me. I am going to summarize the optimization process and purpose in one easy sentence that you need to memorize. Understand this sentence very well and you will understand the whole point of optimization and get on the

fast track to master it. If you fail to grasp the meaning of this sentence, then you will always have losing fights against the Source engine, randomly placing hints and areaportals without knowing really what you are trying to achieve.

Your ultimate goal is to make a specific visleaf "see" the *least* amount of adjacent leaves thus preventing the engine from rendering the content of these "unseen" leaves which will reduce engine overhead and increase frame rate.

That's the essence of optimization in all its simplicity. All the techniques that we will go through in this paper have a unified goal of fulfulling the above statement. Whether it's hints and areaportals, or func_detail and nodraw, it is all about reducing the number of visleaves in the PVS, and reducing the amount of content to be rendered, thus preventing the engine from over-rendering.

If you are interested in going into more details of the BSP method and file format/coding, then these 2 links shall prove to be very useful.

http://www.flipcode.com/archives/Quake_2_BSP_File_Format.shtml

Paper by Max McGuire (07 June 2000)

https://developer.valvesoftware.com/wiki/Source_BSP_File_Format

Valve Developer Community Article

III – Optimization Phases / Systematic Approach

As with any other process, a systematic approach that can be used in every map is highly needed; there is absolutely no point in adding occluders to your map as a first optimization job if your layout, skybox and brushwork are totally un-optimized. The following is a standard sequence that I personally use in my maps and that will help you make your optimization process a fairly straightforward progression.

III.1 - Layout

As surprising as it may seem, optimization starts with the map layout itself. A poor layout can be a hell to optimize while a well-thought layout can pass through with minimal optimization. If your layout is inherently wrongly devised, then all the hints and areaportals in the world won't do you good.

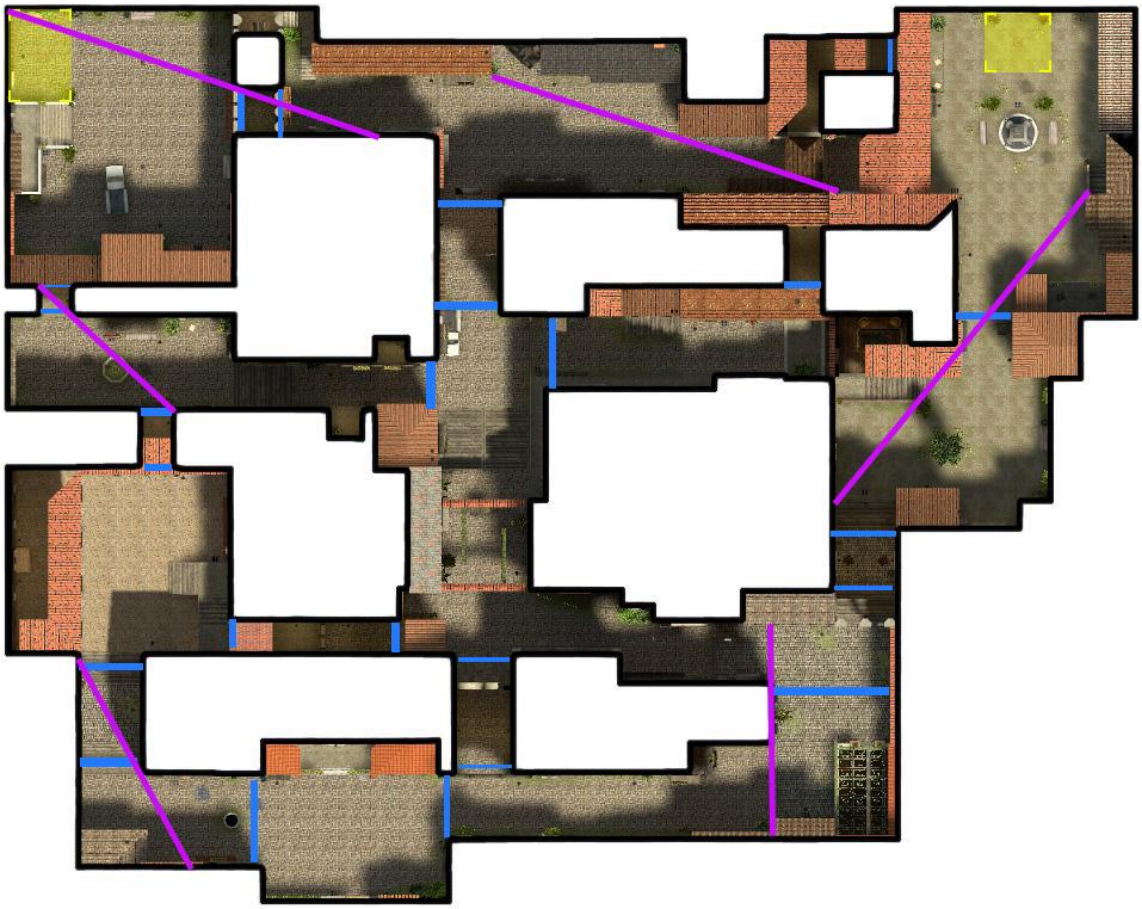
Keep in mind the basics of the Source engine and the bsp discussed earlier in paragraph II to aid you in the layout planning: remember that the bsp is more suited to indoors, rooms and corridors. But what about the outdoors I hear you ask? Well, the outdoors in Source are

“camouflaged” indoors where streets/roads and plazas/arenas replace rooms and corridors on a larger scale, and the same layout techniques apply.

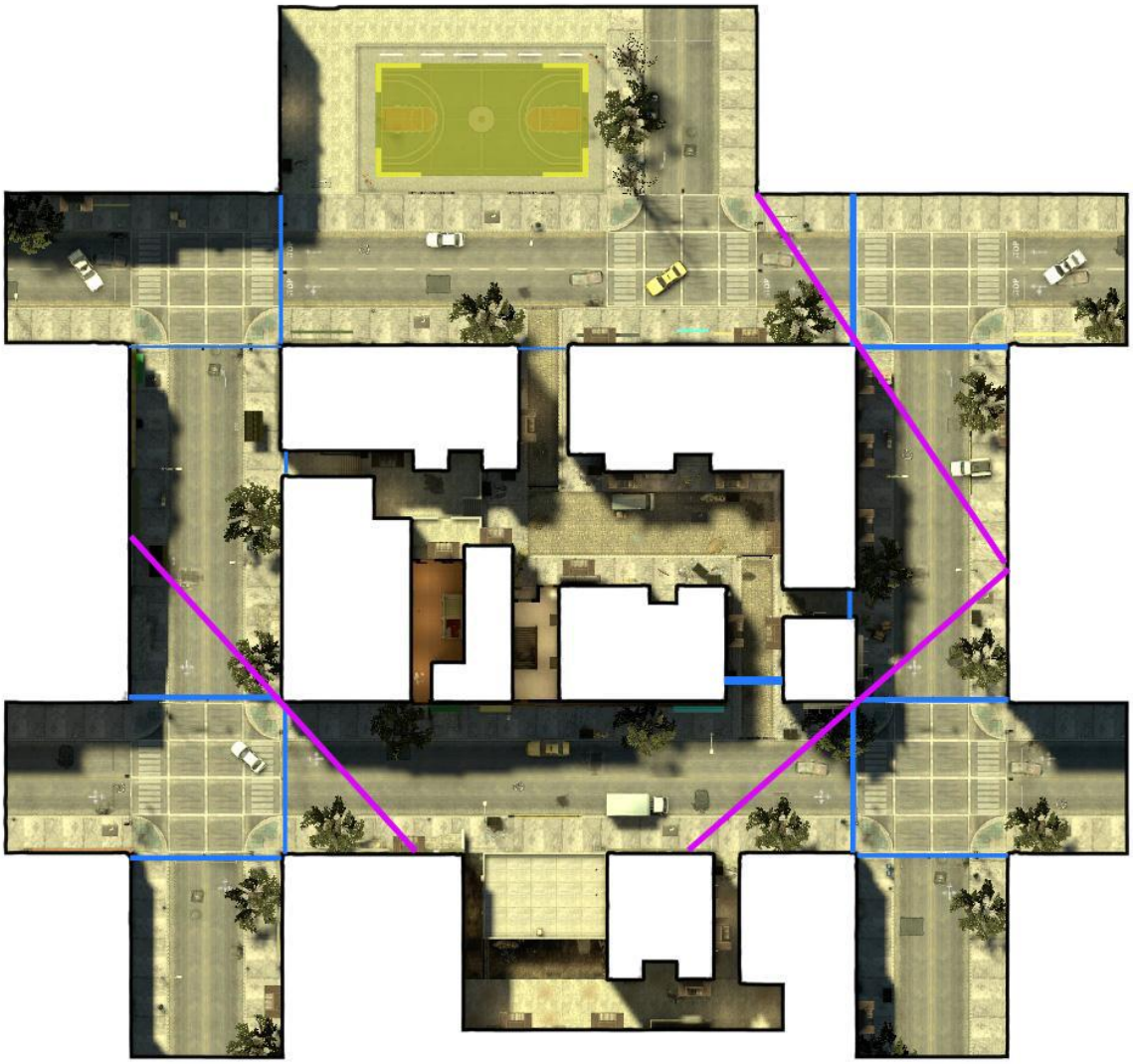
Whenever possible, try to avoid unnecessary open areas with unbroken line of sight (LOS), implement corners to break LOS, add solid structures like buildings or walls to also break LOS. Note that it is possible to have an open-ended map (like de_lake in CSGO) without corners and solid structures, however it will require more optimization and the end frame rate will be lower than a proper “corridor” map.

Your general map layout with corners and properly placed solid structures will greatly help when you reach the phase of hints/areaportals and occluders. In fact, your layout should be laid in a way to help implement these brushes. Let’s have a look at the layout of de_cortona in the next pic (classic figure of 8): blue lines denote areaportals while purple ones are hint brushes. This is not the actual scheme that I used but it should give you an idea how the layout itself is laid to help implement hints and areaportals at a later stage.

Corners are very useful to have those angular 45° hints to eliminate direct line of sight between the opposite sides of the corner. Areaportals are generally placed at hallway/corridor ends to delimit and separate areas (don’t worry, hints and areaportals will be explained later; for now, just focus on their placement and its relation to the layout).



Another map with more open-ended layout and gameplay: cs_east_borough.



Remember when I told that outdoors in Source are just “concealed” indoors? Here’s your proof. The map is about open city streets and outdoor cityscape. However, if you look closely at the layout, you would see that streets are nothing more than large-scale corridors with corners and the same principle applies. Hints are applied on corners (and many other places not shown here) while areaportals are used to separate areas at each street’s end.

To go into details about layout planning and sketching, I recommend reading my previous technical paper titled **“Planning to Win Sketching Your Level”** which can be found here http://source.gamebanana.com/tuts/10980.

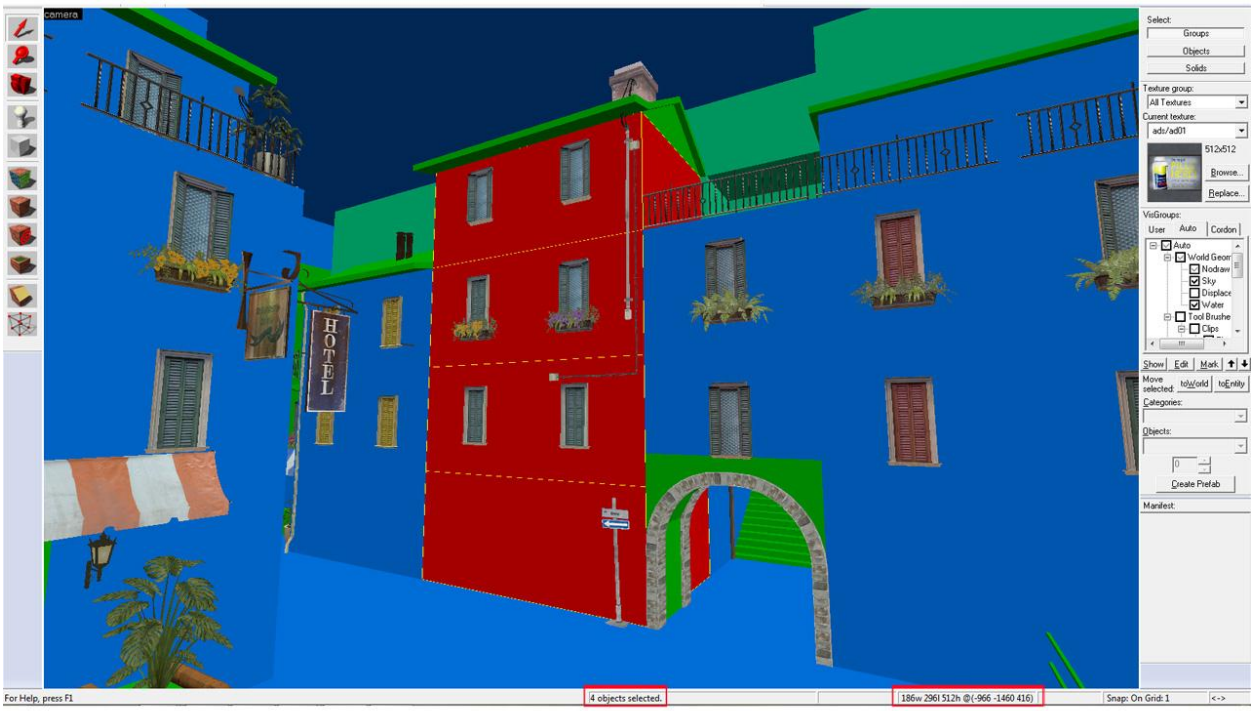
III.2 - Brushwork

With the layout clearly planned and sketched, it is time to start building your map and here is the second cornerstone of your total optimization: optimized brushwork.

If you carelessly build your brushes, it will backfire at you in the end when you reach the hints/areaportals phase; you will have a hard time aligning these brushes to your messy brushwork and they will not be easily placed as I showed you in the 2 previous layouts.

Get in the habit of building your brushwork in standard sizes, mainly power of 2. It is always a good idea to have big brushes in 128,256 and 512 units' size while medium and small brushes should be in 8,16,32,48 and 64 units. Not only you would have an easier time to optimize a standard block-size map, the compiler will have an easier time during vvis calculation and the compiler itself will have easier time to figure out the visleaves even before your intervention to add hints and areaportals.

Make sure "snap to grid" is enabled in Hammer and keep the grid size on 8 for blocking the map with backbone base brushes then go down to 1 when it comes to details and fine tuning. As an example, a building of 512 units high can be made of 4 blocks of 128 units each. This will allow the engine to cut the visleaf horizontally and will give you greater flexibility for texture application.



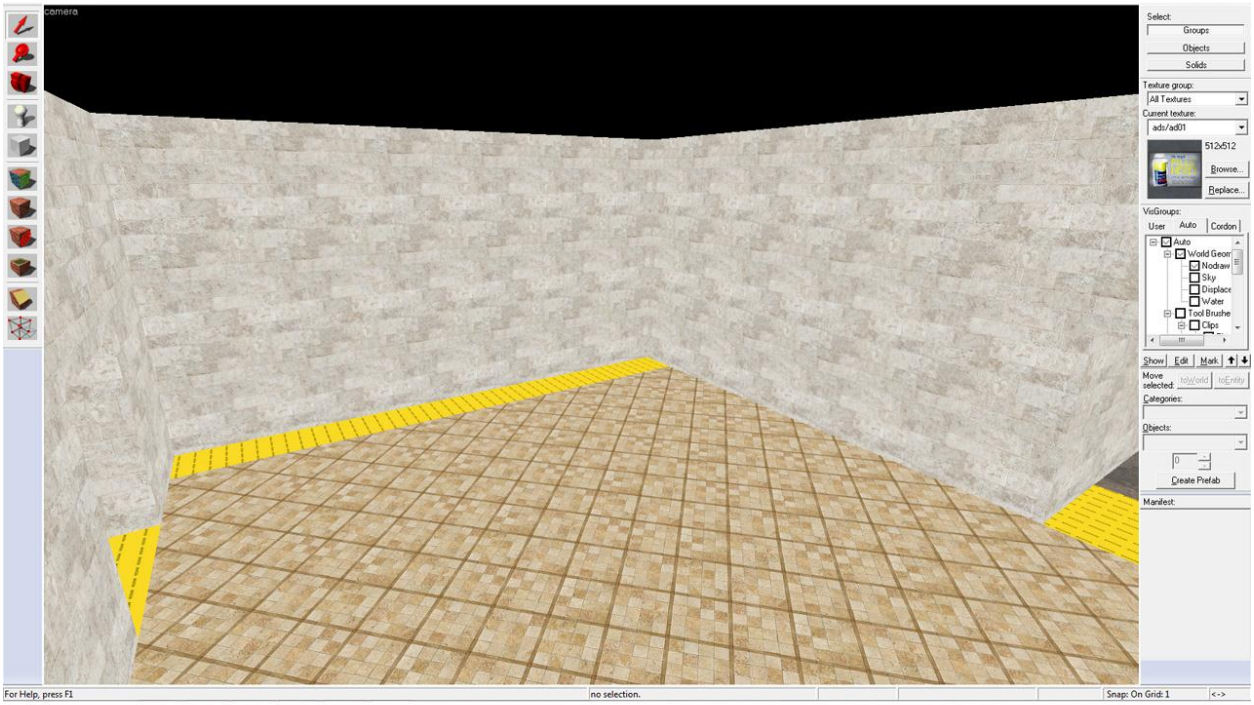
The above is a screenshot from de_spezia_pro (Hammer 3D flat view)

I highlighted the brushes of this building in red as well as the total building height (512). You can see it is made of 4 brushes of 128 units high each; this way, even if you forget to add horizontal hints later, the compile tools will figure out to include a visleaf cut at each of the brushes' intersection.

The last thing that you need to take care of with brushwork is to try to have your brushes as square as possible, when it comes to the backbone brushes of your level (the big regular world brushes that are used to block visibility, to seal the level and support visleaves cuts). Feel free to have the weirdest and most creative shapes that you can come up with (within the limits of the engine of course), when it comes to decorations and detail brushes. These brushes will be turned to func_detail as we will see later and won't affect visibility calculations or the time it takes vvis to do these calculations.

Whenever you have odd-shaped brushes or curves, think immediately of turning them to details and keep your base brushes as square (or rectangle) as possible.

This is bombsite B in de_spezia_pro viewed in Hammer. You can see a lot of circular, angular and odd-shaped detail brushes and displacements. What happens if we hide everything in Hammer keeping only the world brushes visgroup?



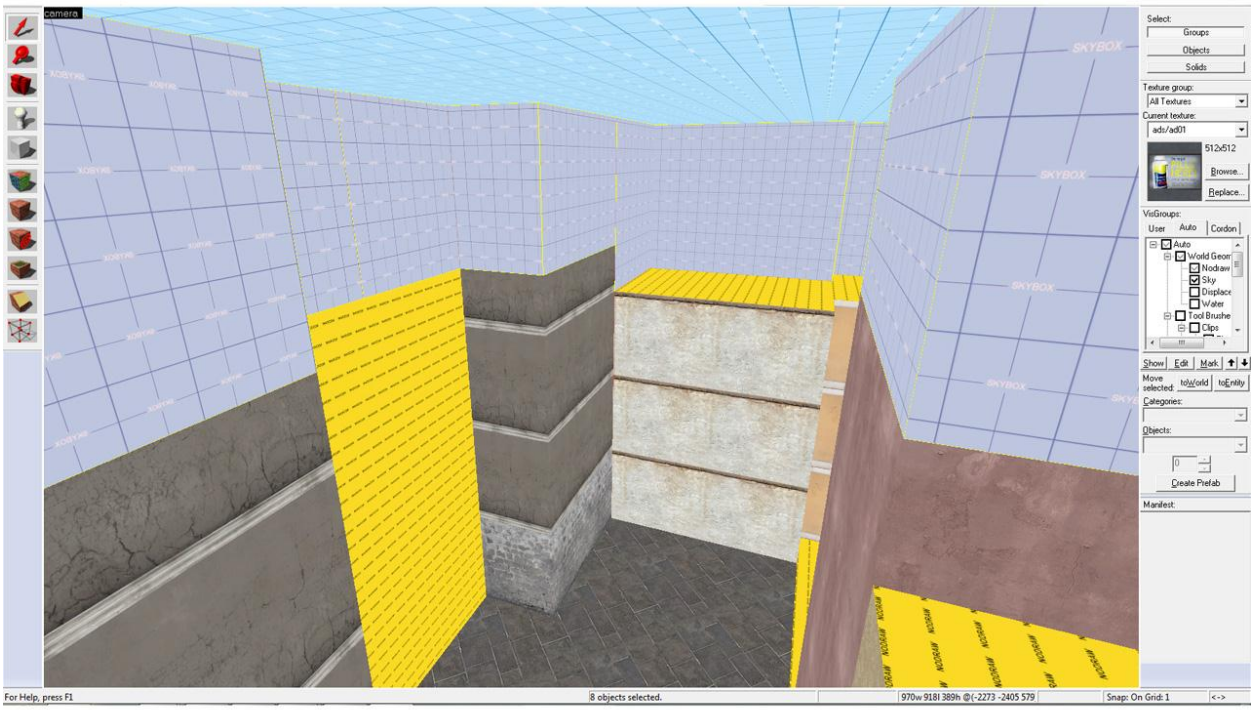
Amazing, isn't it? The core of bombsite B is nothing more than…a flat square (4 walls and a floor). This will simplify visleaves' creation & division to the maximum allowing vvis to compile in a couple of seconds, and ensuring some luxurious fps in-game.

III.3 - Skybox

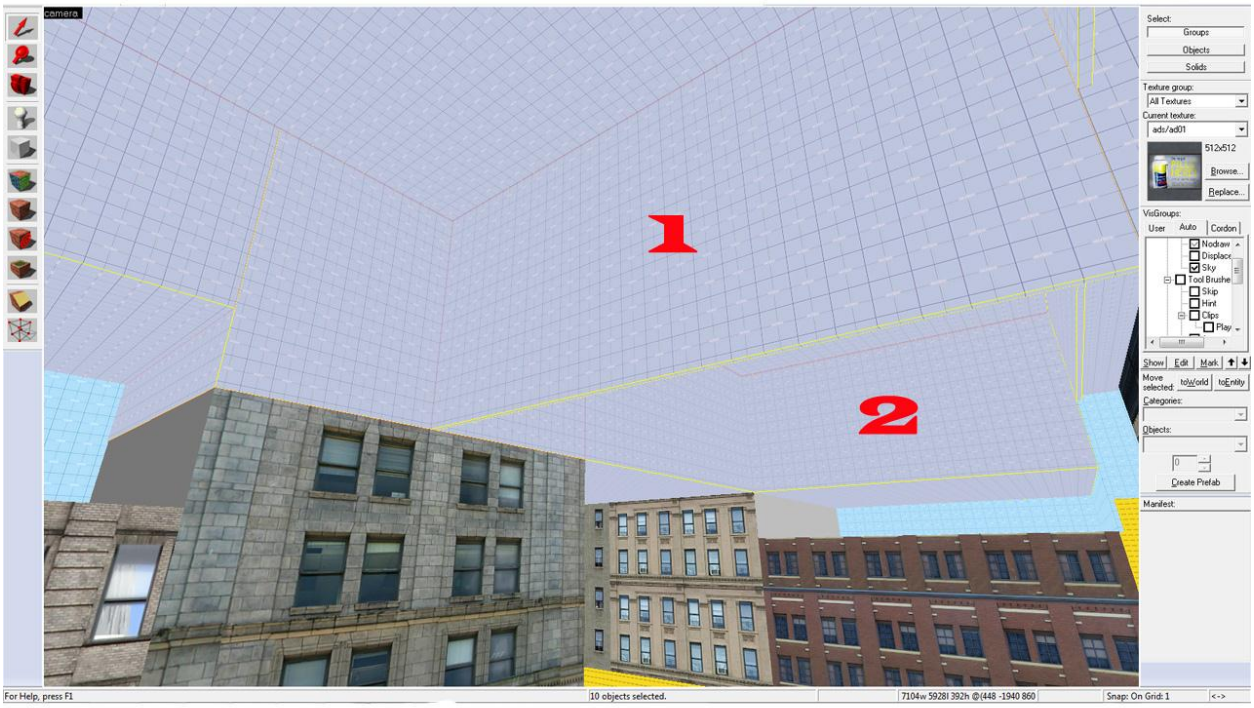
Let me start here by being very blunt: Never, ever envelop your map with a big hollow skybox cube.

While you might "get away" with this technique if your map is a very small deathmatch-style map featuring a small open yard with couple of crates, you will literally kill your level if it is of fairly decent size: vvis might crash during compile or take ridiculously long time to finish, and even if the map manages to compile, you will have horrible frame rate in-game in addition to an assortment of bugs and glitches rendering the map practically unplayable; in other words, dead.

Always follow the contour of your map as closely as possible when building your skybox brushes. Another thing to keep in mind is to minimize wasted space between the map's brushwork and the skybox; this can be done by inserting additional skybox brushes to fill the gaps. Not only this will eliminate the generation of additional visleaves, it will also help in delimiting areas, blocking visibility and creating new locations to implement hints and areaportals.



This is the main street in de_spezia_pro; I already highlighted the skybox brushes to showcase how they neatly trace the contour/edges of the underlying buildings. Also notice that there is no gap between the building top and the horizontal skybox brush as mentioned earlier to eliminate unneeded visleaves. This might not be possible on all buildings especially if there are props/detail brushes on the roof that would get hidden if you add a skybox brush around them, but high buildings with no/few items on top are prime candidates for a skybox brush on top of them.



Here's another screenshot from cs_east_borough in Hammer. The skybox brush labeled 1 is added/used to delimit 2 areas and separate streets using an areaportal that would go from ground level to the bottom of this brush. Skybox labeled 2 is a further example of what I already mentioned in the previous example. You can see here that this brush does not touch the building beneath it, otherwise, the building's wall to the left will be hidden in-game; but still, this brush will eliminate wasted space above the building and will be the foundation for another areaportal, perpendicular to the one made under brush 1.

It goes without saying that when you are building your skybox, you need to hide all visgroups in Hammer except for world geometry (regular world brushes) to make sure that your skybox will be neatly complementing your world brushes. You can unhide other visgroups after you build the skybox just to make sure that no important elements (props/func_details, particles…) are hidden by a part of the skybox. If this is the case, then you need to modify this specific part of your sky to make sure all elements are visible in-game to players.

One last pitfall to avoid is making the sky too low or too close to players. This will hinder the motion of physics objects as well as thrown grenades, and will break the immersion and realism in your level. Always keep the skybox at a safe distance from the player to allow for such maneuvers.

III.4 – No draw

As we have seen in paragraph II, the source engine carves space into convex visleaves, and automatically separates the inside of the level from its outside. The brush faces that touch the "void" outside your map will be culled (removed) by the engine whereas the faces that are inside your level but can't be seen by the player will still be drawn and rendered thus putting additional load on the engine.

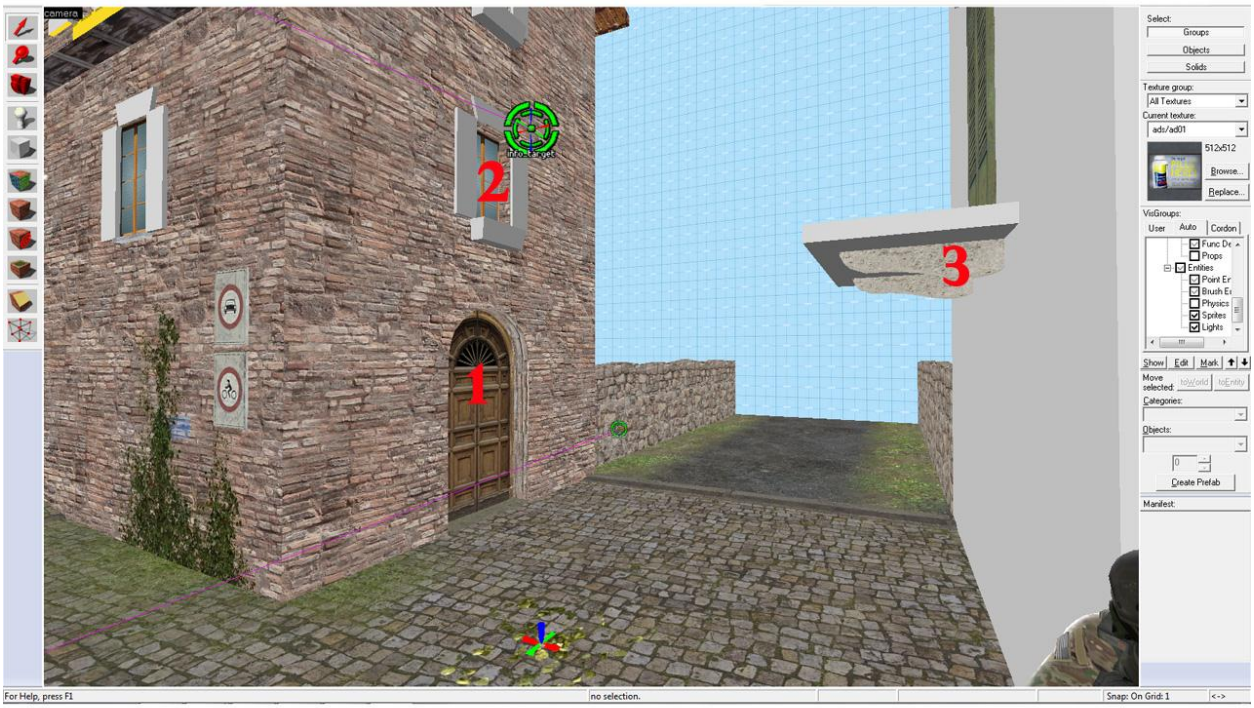
What can we do in this case to force the engine not to render certain faces that are unseen during normal gameplay? Enter the "nodraw" texture. As its name suggests, it instructs the engine to remove the face that has the "tools/toolsnodraw" texture applied on. Not only this face becomes invisible in-game, the engine will not draw it further reducing polygon count and overhead; in other words, higher fps.

I can't stress this enough: apply the "nodraw" texture to any surface that the player won't see during normal gameplay (I'm not including observer cam or by using noclip). Don't forget faces from the big roofs to the tiny rails on a balcony as every nodraw will help improve performance.

The best candidates for nodraw are usually inaccessible rooftops, the backside of an inaccessible building, and the backside of a wall that the player can't go behind and so on. I see many mappers forgetting that nodraw can be partial and angular too. If you have a complex brush where some faces can't be seen from a certain angle, then these faces are nodraw candidates; you don't need to nodraw the full brush, just the faces that can't be seen.

I can hear you wondering now: but that's a whole lot of faces to manage; what if I forget some? Well, the solution couldn't be easier. Apply the nodraw on the whole brush then only texture the faces that will show during gameplay. This way, you will be sure that no face is left unnecessarily textured.

Consider the following 2 screenshots from de_cortona (Hammer view)



This shot is taken from CT spawn and players cannot go through this road; they can only reach up to the decal on the floor (there is an APC and a clip brush to prevent them for going further but I removed them in the pic for the sake of clarity). This means that the player will have exactly this view once in-game.

I already labeled the door with arch as 1, the window/sill/flower bed as 2 and the balcony/support as 3. You might be thinking: where do I need to apply the nodraw other than the back of the building? Check the 2nd screenshot and you will be surprised.

That's the same location; however, the screenshot is taken from the opposite side of the road where the player has no access in-game. I also labeled the same places with the same numbers as in the previous pic so you can compare.

You can see that I nodrawed half the arch surrounding the door and all those tiny edges on the window as well as the edges of the balcony support and door. Naturally, the building back (labeled 4) is obviously nodrawed. Some of you might be thinking that this may be too much detail to nodraw; trust me when I tell you that every nodraw counts and you will be thankful and at the same time rewarded, when you see the high fps in your level.

III.5 – Func_detail / Displacements

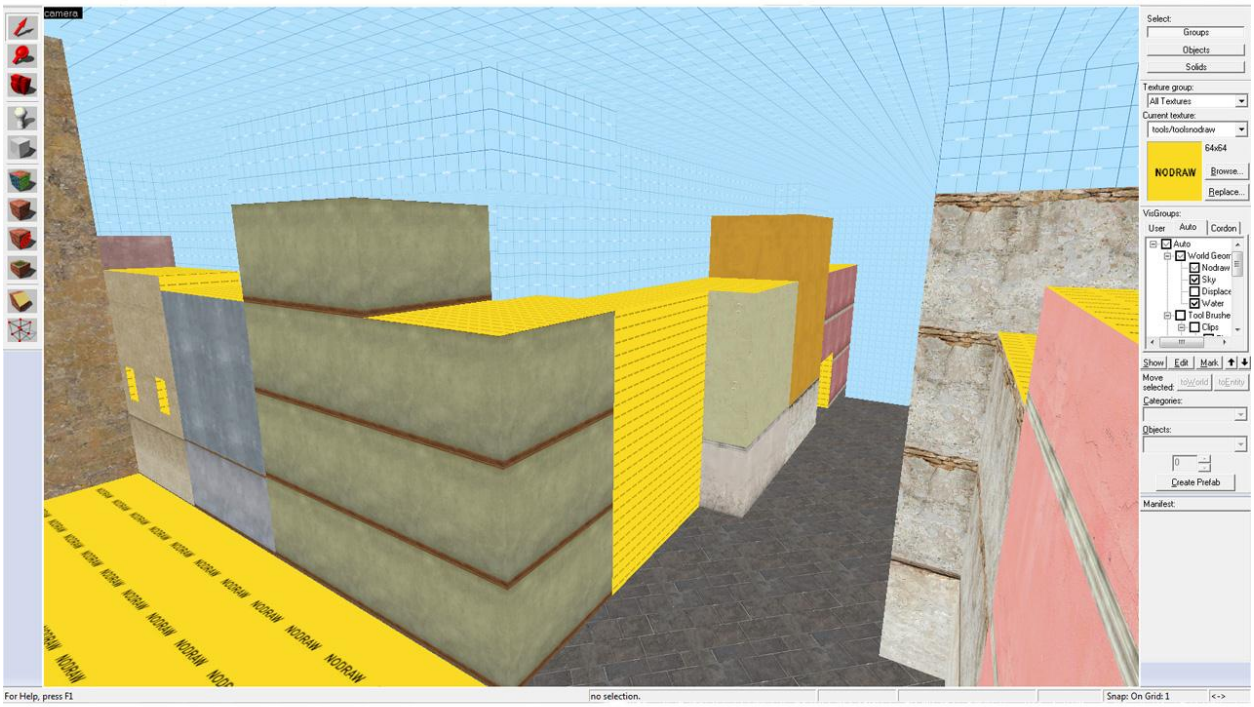
If you have been paying attention in paragraph III.2, you would have deduced that you need to use func details as much as possible. Anything that does not constitute the backbone of your level like buildings and floor and anything that is not used to break LOS should be switched to func detail.

Don't get alarmed if large portions of the map become func details as long as the level is properly sealed with regular brushes (func details, displacements and props don't seal the level and will cause a leak). Func details won't create new visleaves thus reducing the load on the engine; vvis will have an easier time calculating visibility and your PVS will be a lot cleaner and less complicated.

In addition, func details are internal entities that get “consumed” during compilation. This means that they don’t exist as an entity during gameplay further reducing engine overhead (just like prop_static).

This is the main street in de_spezia_pro overlooking the beach and CT spawn. You can see a lot of detail here: arched roof, angled roofs, roof edges, door, trims, windows, balconies…

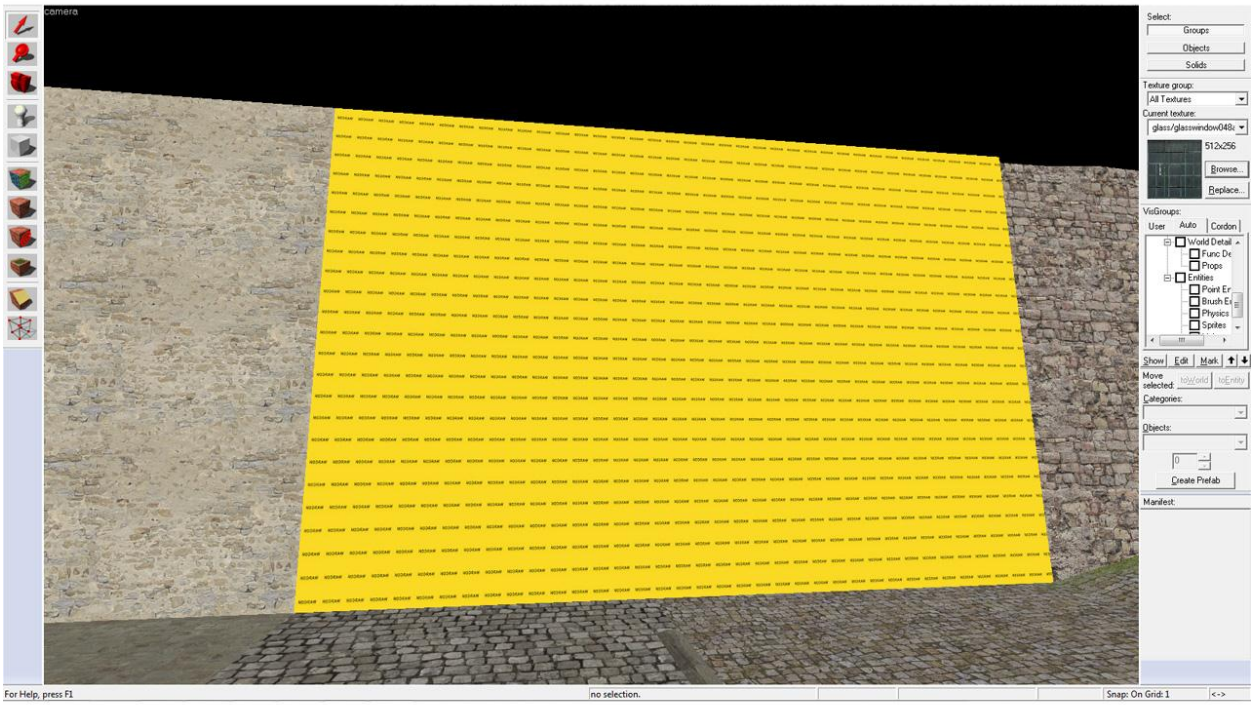
What happens if we hide all detail visgroups (mainly func_detail, static props and displacements)?



Just like bombsite B in paragraph III.2, all what is left are some perfectly flat and square (more or less) brushes. That is why I told you earlier not to panic if most of your level becomes func_detail just as long as the backbone of the level is there to seal your map.

Since I am a nice guy, I will show you another trick to optimize your brushwork using func_detail. Let's take a look at this screenshot from de_cortona.

Those are some fancy decorations on that building, aren't they! You can see arches, encased doors and windows and you already know that keeping these as regular world brushes is a big no-no. The trick here is to make this façade out of a thin brush (16 units), turning it to func_detail while keeping the underlying brush as a nodrawed regular world brush.



Here you go. After hiding details, we are only left with a nodrawed flat brush, and as said before, this will have tremendous effects on your vvis time and your in-game frame rate.

This is another screenshot taken from the side to make the above clearer.

The brush labeled 2 is the world brush from the above screenshot acting as backbone to seal the level and as a support brush for the skybox and hints/areaportals. You can see that the thin brush labeled 1 is actually several func_detail brushes to make it easier in manipulation and texture application.

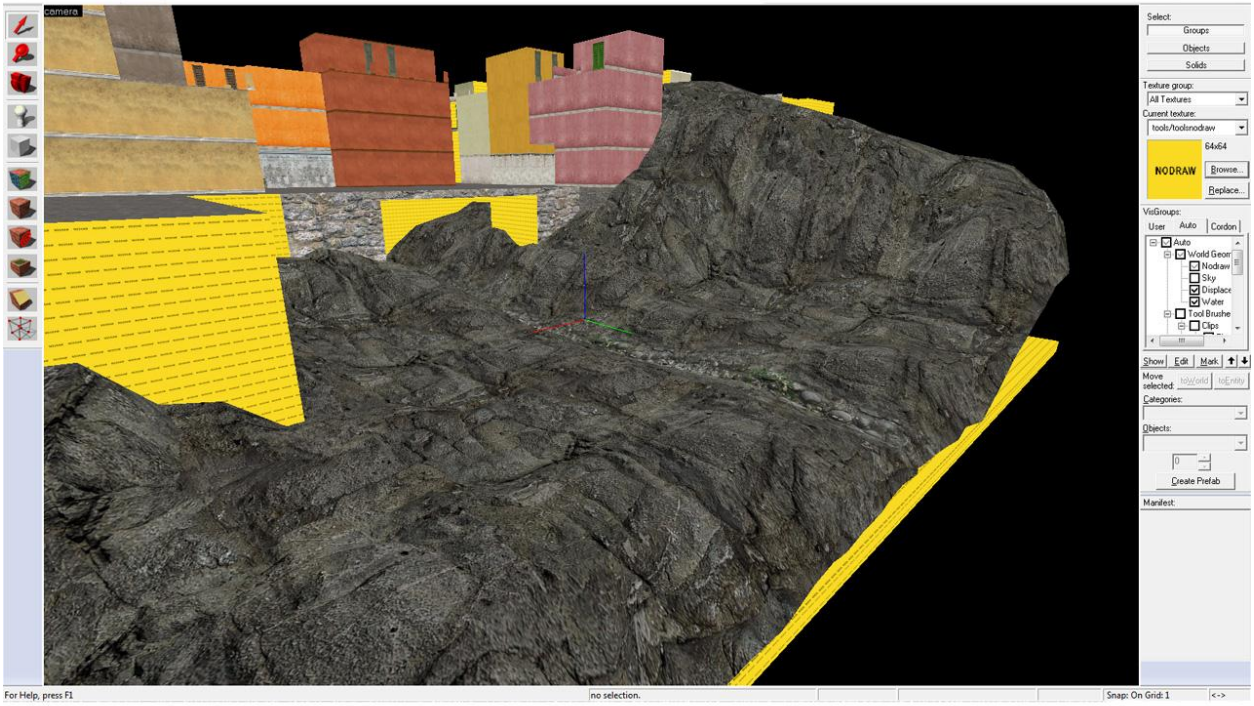
Now that we established the importance of func_detail in your optimization plan, I still need to inform you of one small annoyance (if you haven't figured it out by now, Source engine has lots of small annoyances ☺). Despite not counting towards visibility and visleaves' calculations, func_details still count towards your total brush count. If your map is large and detailed, then you might hit the brushwork limit (8192 brushes and 65536 brushsides).

In this case, it is always safe to turn to props to replace your brushes. However, in some cases, you can't find an adequate prop or you don't have the time or skill to create one; are we stuck then? Not at all.

Enter displacements; introduced with the Source engine, they are dirt cheap when it comes to sucking engine resources (the engine handles them pretty well). To top this, and unlike brushes where you are limited to clipping and vertex manipulation, displacements can be organically sculpted in the 3D view of Hammer, thus enabling you to have fluid, non-edgy shapes to wow your audience.

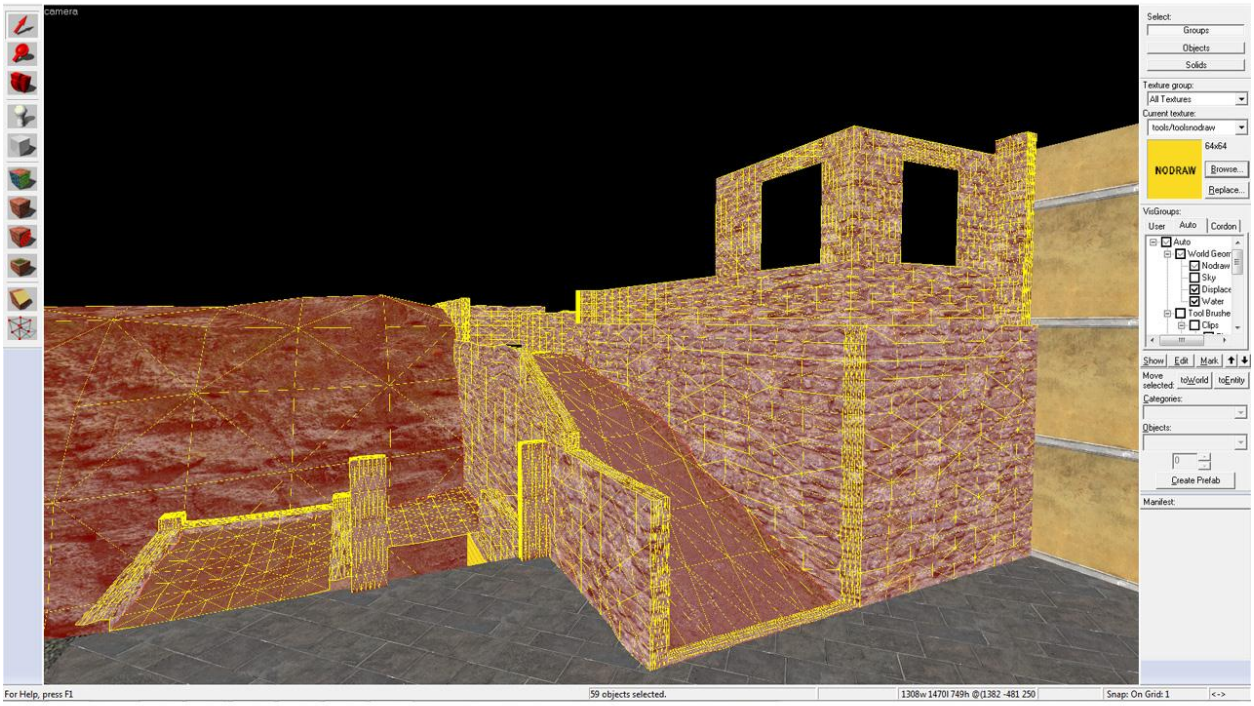
Keep in mind that displacements cannot be turned to func_detail since they also do not block visibility or create visleaves themselves, and they cannot have the nodraw texture since the engine will cull the faces not turned into a displacement.

In addition to using them mostly for organic shapes (mountains, rocks, roads…), displacements can be perfectly used as square, edgy shapes to replace func_detail and reduce brushwork count in your level.

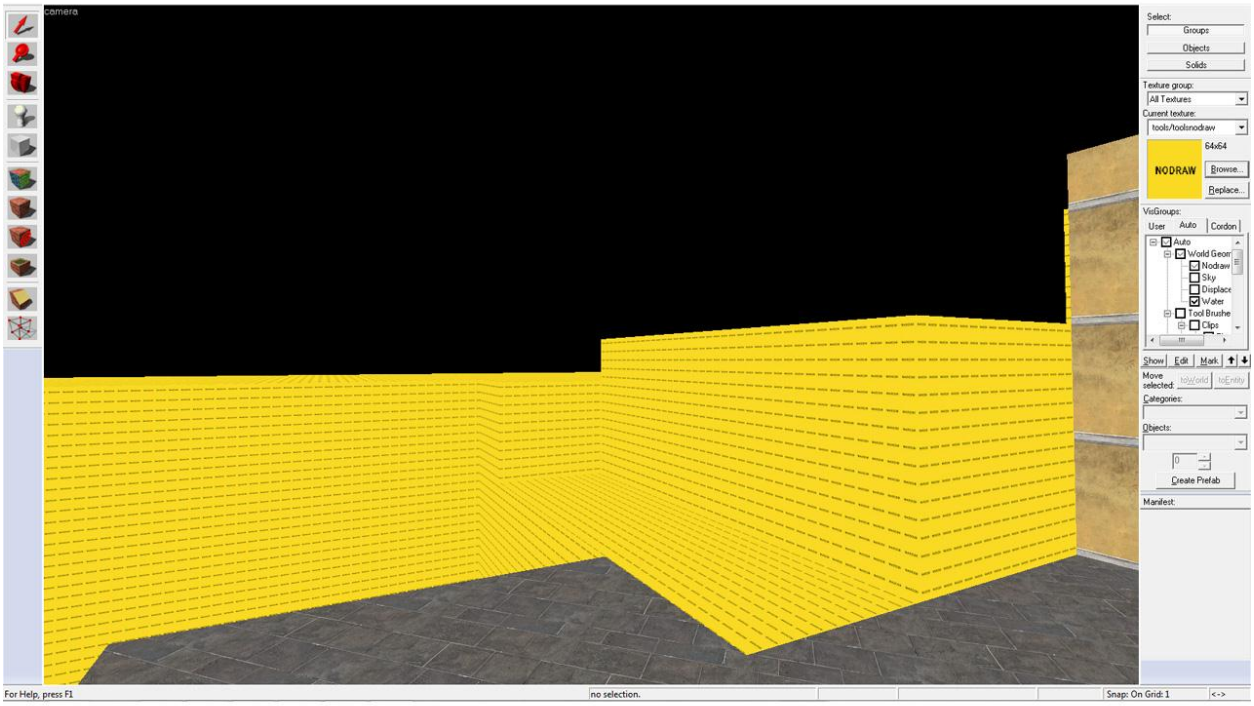


A prime example of organic shapes is the beach near CT spawn in de_spezia_pro. It's entirely made of displacements and is very cheap to render despite its big size and its wide coverage area. Also note the nodraw brush underneath it (it is connected with the skybox brush which is hidden here for clarity reason); it is needed to seal the level and prevent leaks, since displacements don't seal levels as I already told you earlier.

Here's another example from the same map where an entire structure (old ruined fortress) is made of displacements instead of func_detail brushes.



You can see that the ruined fortress, its walls, the road, the small garden, the gate support walls and the cliff in the background, are all made of displacements. If I hide them (next pic), I am left with a couple of square nodraw brushes to seal the level.



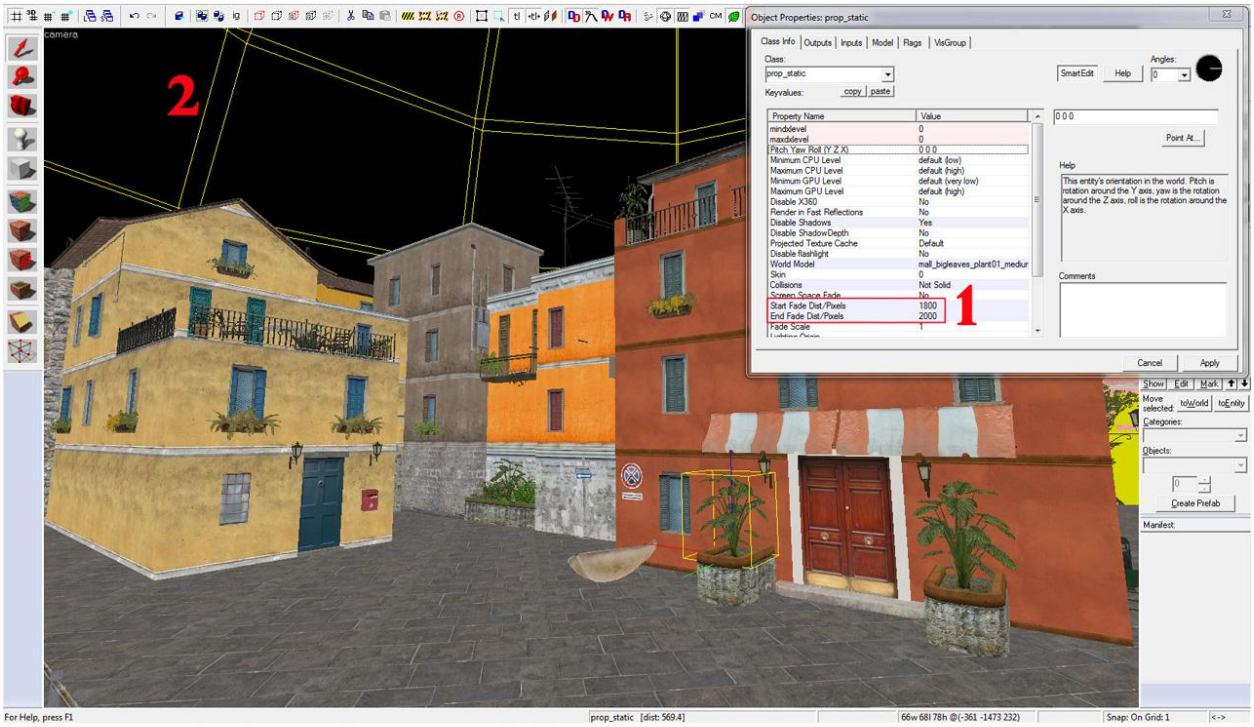
And as I mentioned, displacements can be used to replace blocky brushes. Here's a fine example. These doors are converted to displacements to reduce total brush count (remember to only convert the faces that you need, not the whole brush). The same principle of nodraw applies here: the faces that the player will not see shouldn't be turned to displacements. If by mistake, you convert the whole brush into displacements, you can still select the displacement in the 3D view, go to its properties and destroy the faces that you do not need.

These doors are converted to displacements without any sculpting, just to decrease the brush count in the map.

III.6 – Props fade distance

As we have seen in the previous paragraph, props, more specifically prop_static, can be a great alternative to using brushes. Prop_statics are internal entities not counting towards total entity count during gameplay and can be optimized themselves during their creation (in a modeling software outside hammer) by using level of details (LOD). However, having too much on screen props showing at the same time can negatively affect your fps.

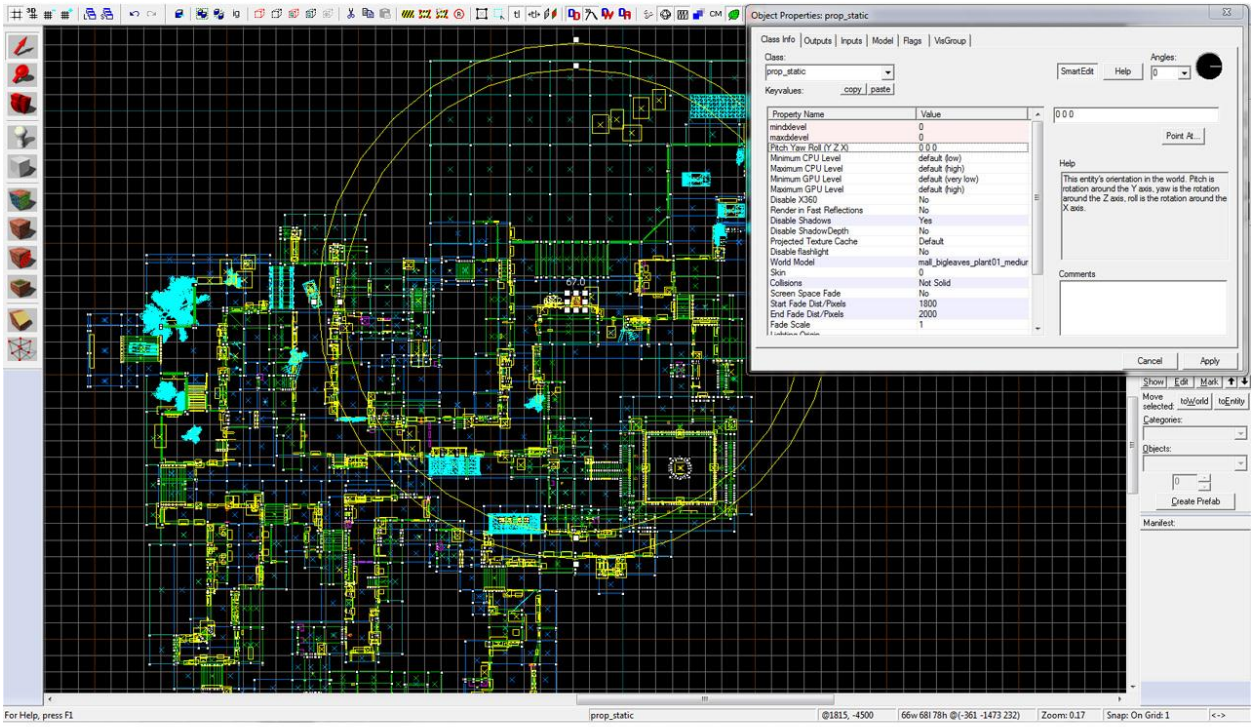
What can be done in this case, you might be asking. Props can be optimized by setting their fade distance in their properties. Each prop has a start fade distance and an end fade distance. You can specify those numbers in the prop's properties window.



The highlighted rectangle labeled 1 is where you input the fade distance in Hammer units. These numbers mean that this prop will start fading at a distance of 1800 units from the player and will disappear completely at a distance of 2000 units; a prop that is not rendered will reduce engine overhead and increase fps.

The yellow lines labeled 2 are actually the boundaries of two 3D spheres showing the start/end fade distances in space (the fade distance is the radius of the sphere). These will help you visualize the distances and determine the corresponding values: you can tweak the numbers and immediately see their effect with this bounding sphere, so you can decide of the accuracy of your set distances.

The same can be seen in the 2D view of Hammer.



Since this is 2D, instead of spheres, you get here 2 flat circles showcasing the corresponding fade distances (again the radius of the circle is the fade distance). As a rule of thumb, always have the end fade distance to be higher than the start distance by at least 200 units; this will ensure a smooth transition while fading, otherwise, the fade effect will be abrupt and unrealistic in-game.

Care must be taken not to fade critical props too soon though as this can impede gameplay. These props will be used by players for cover, so it's important for these props not to disappear from a distance, otherwise the player taking cover behind them will be exposed to the enemy players. Needless to say that the element of surprise will be lost and the player will be really pissed off. On the other hand, small props are a prime candidate for fading such as flower pots/beds, small bushes, trash bins, light bulbs, fire hydrants, lamp posts since most of them are non-critical to the gameplay and probably won't be noticed by the player when he is at certain distance from them.

On to examples since we all love colorful screenshots:

The middle street in de_cortona is of utmost importance since it connects CT to T spawn; however, both teams should not be able to have a free line of sight, otherwise snipers will ruin gameplay. You can see that I added a stack of crates in the middle (red highlight) and this is exactly the definition of a critical prop. To apply what we have discussed earlier, the fade distance of this prop should be equal or higher to the distance from either CT or T spawn to this prop. In this way, players along this street will always have this crate visible, which is critical for a balanced gameplay.

Another example from cs_east_borough.

You can see here 2 cardboard boxes (red highlight) containing washing machines that are used as cover when sniping across the street. The fade distance for this prop should be equal or higher than the total length of the street, so players at the street's end can still see this prop (which in turn will keep hiding whatever players behind it). If the street is 2600 units long, then set the start fade to 2600 and the end fade to 2800 units. You can also notice some newspaper stands, fire hydrants, signs, bottles and other small props. Those can be faded much earlier than the boxes, since they won't be used for cover and are aesthetic elements to embellish the visuals.

III.7 – Hints

We have just reached optimization point number 7 out of 10; this tells you that almost 70% of your optimization is done before even touching hints and areaportals.

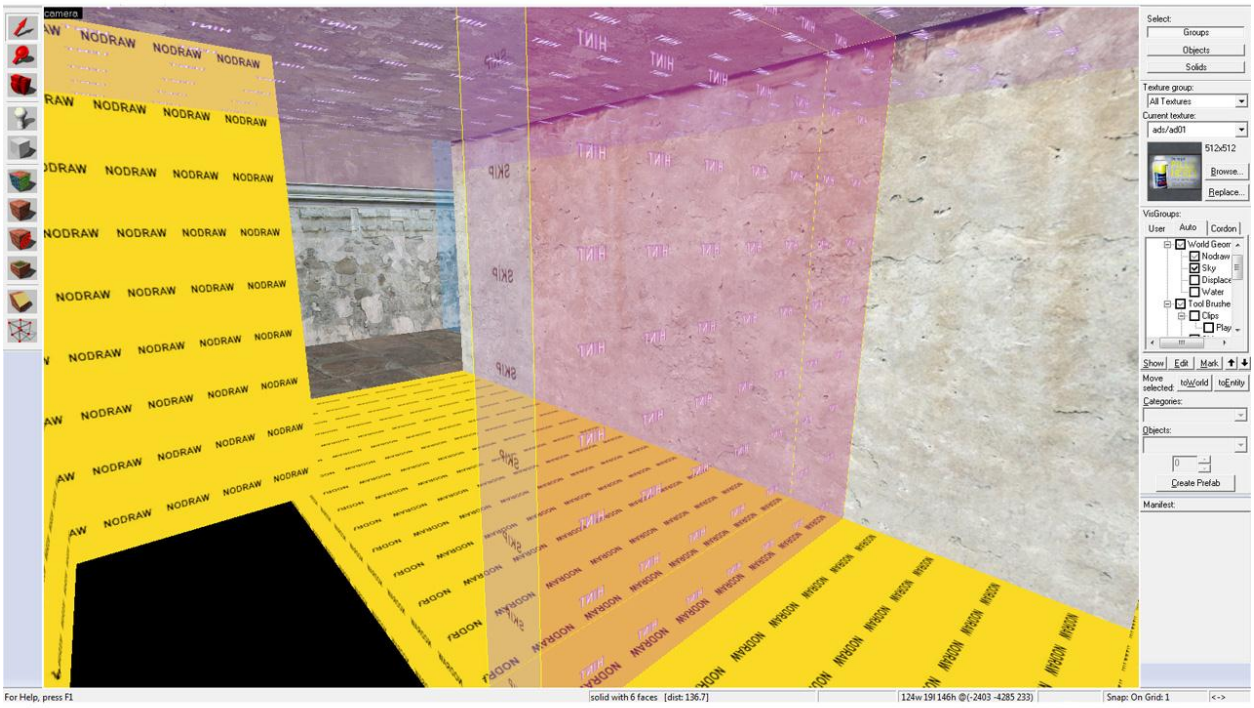
In de_cortona's case, when I reached the hints/areaportals phase, the map already had high fps all around; this is just to tell you that hints/areaportals/occluders should be your last weapon in further improving your fps and not some sort of magical tools that will make your wrongly-laid, poorly-built map, smooth and lag-free.

Hints are very useful and easy to use to control visibility and visleaves' divisions. They have been around since the days of Quake 2 as far as I can remember. I was introduced to them back in 2000 by Chris 'autolycus' Bokitch who was running the VERC website (the grandfather of the current Valve Developer Community website). Things were different back then and tutorials were scarce, and Autolycus (later hired by Valve) was our only resource for Half-Life

related tutorials. I read the hint tutorial several times to fully grasp it but the only way that made me master the subject was experimenting with test maps, which is what I recommend you do after finishing reading this tutorial.

Alright, with that cleared, we can move on to see what are those hint brushes, what do they do and why are they called so (I personally disagree with the name ☺).

Hints are regular world brushes with the "tools/toolshint" texture on 1 or more faces and "tools/toolsskip" texture on the remaining faces.



As you can see, it's a regular world brush, not an entity. The surface that is facing us has the hint texture while the other faces have the skip texture. The engine will cut the visleaf and separate it into 2 only on the hint face; the other skip faces are…well, skipped ☺ (the engine will simply delete these faces in-game).

It's called hint because you are actually giving the engine a hint to cut the visleaf exactly along the "hint" face, and here is where I disagree with the name. As it is now, the name implies that you are merely giving the engine a subtle hint of where you think the visleaves should be divided; it also implies that the engine might accept your hint or it may simply reject it. In reality, the engine will blindly obey and carry out your request, and will cut the visleaf exactly where you put your hint face on the brush. For me, it's more like an "order brush" rather than a hint brush ☺.

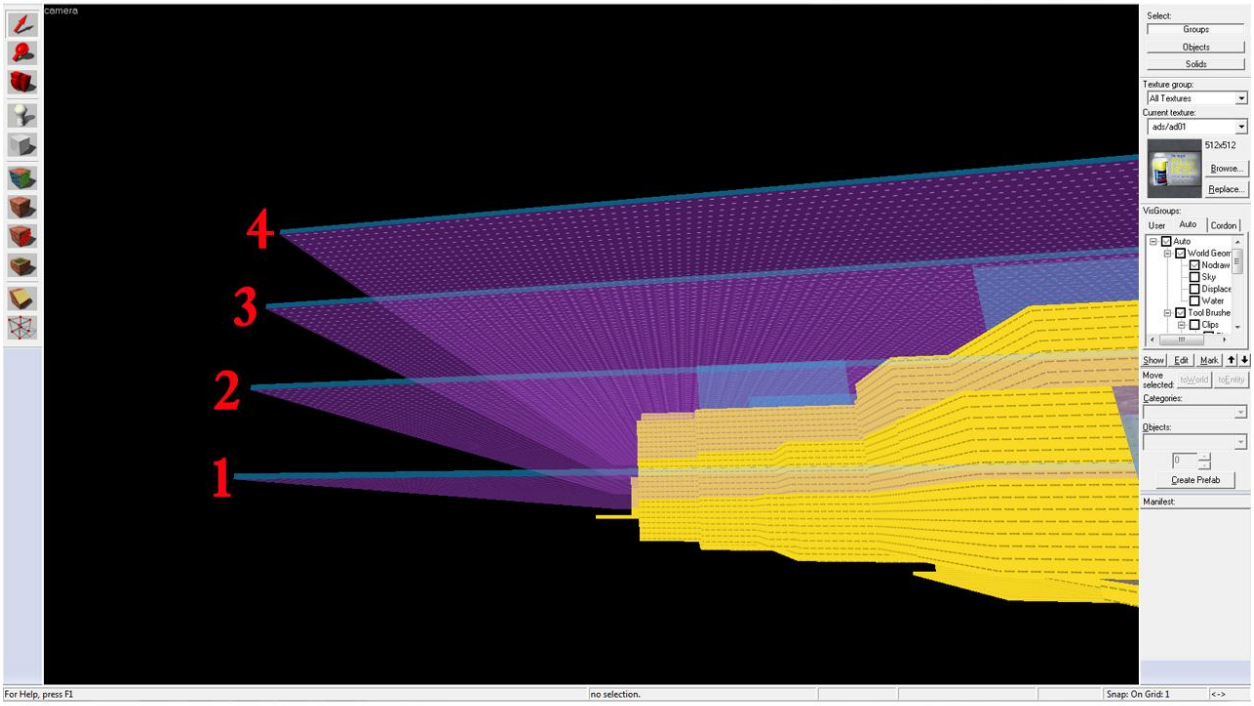
Now with that taken of my chest, let's see how and where we should implement these order…sorry, hint brushes.

First and foremost, you need to hide all visgroups in Hammer except for the skybox and regular world geometry (and the hint group of course); this is to make sure that your hints will be correctly placed between brushes that are actually taken into consideration for visibility and visleaves calculation. In addition, keeping other groups will clutter your view and make it impossible to know where to insert your hints.

As a start, it is always wise to add at least 2 horizontal hint brushes for open-ended maps (one at 128 units, another at 256 units and more can be added as necessary) to prevent having tall visleaves that "see" each other from across the map ("see" here means having a direct LOS).

I usually have 4-5 horizontal hints in my maps since they are large and fairly open: I add the first one at 128 units high (hint face down, skip on all other faces), then clone this brush at 256, 384, 512 and more if skybox allows for it. If your map is mostly indoors with lots of rooms and corridors (underground base for example), then these hints might not be necessary, but they are much needed for all other maps that feature outdoors.

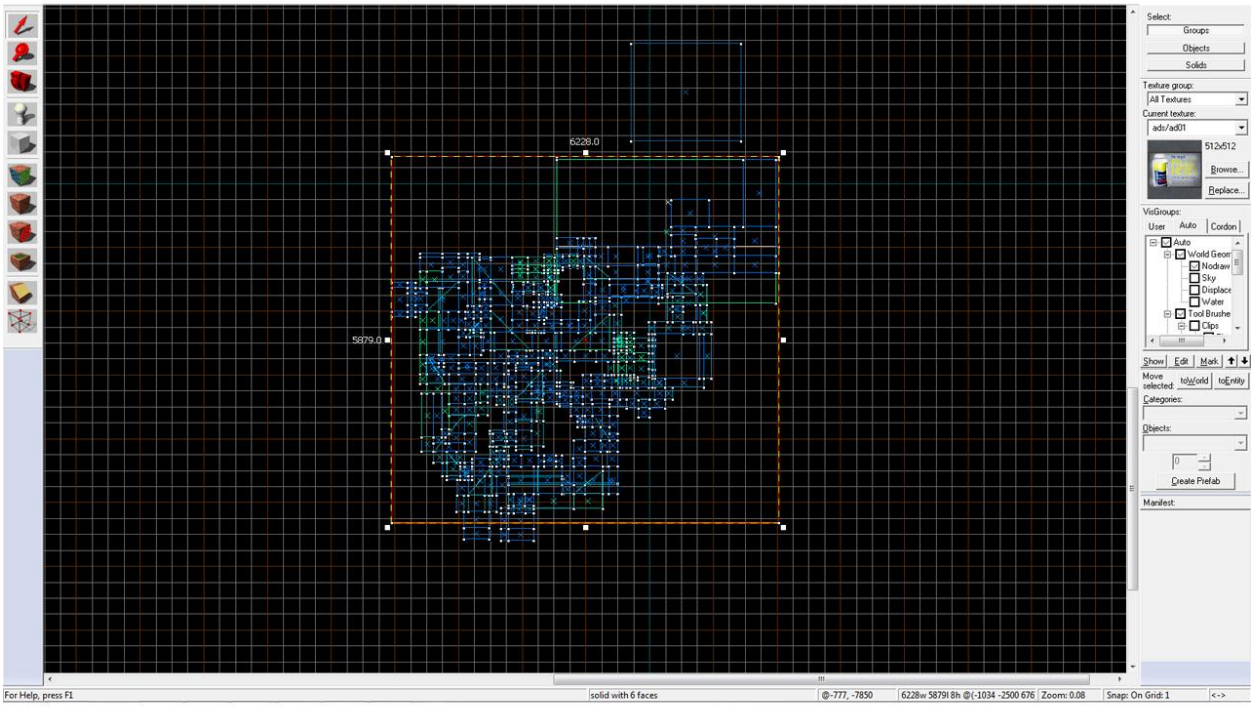
Let's see these horizontal brushes in de_spezia_pro.



I numbered them from bottom to top starting with the one at 128 units and ending with the 512 one. You can notice the purple hint face on the bottom of each brush (facing down) while all other faces are cyan (light blue) indicating the skip texture.

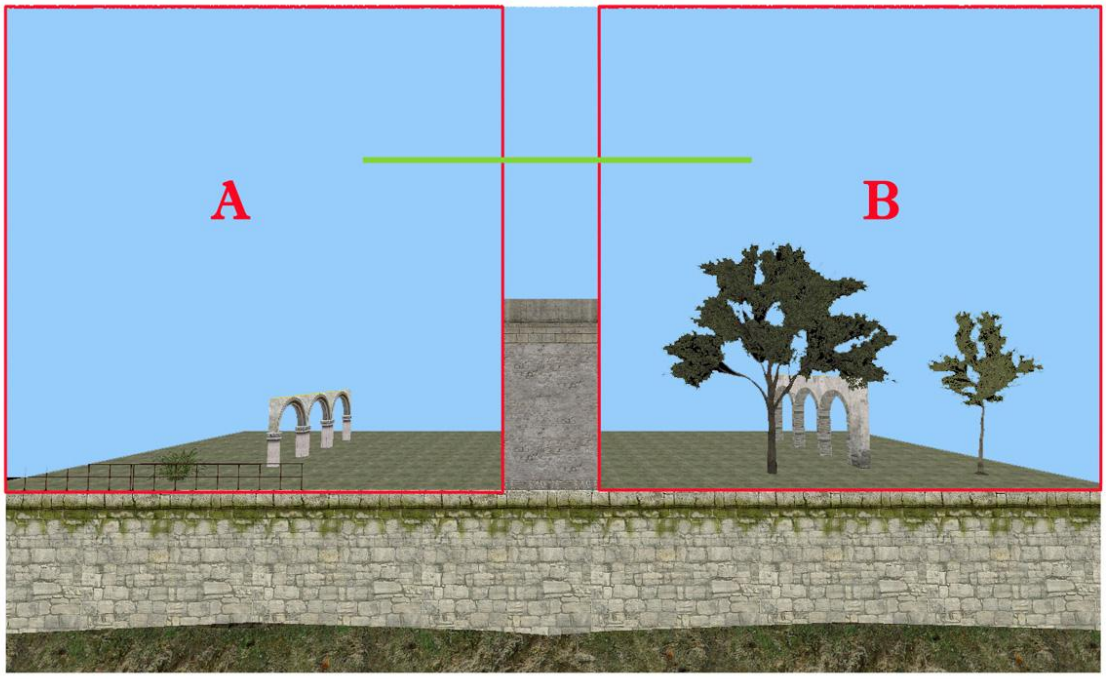
If you followed my instructions closely in paragraph III.2 about building your brushwork in standard sizes, you would immediately notice that these horizontal hints will perfectly match your world brushes at 128, 256, 384, and 512 units' high; your map geometry couldn't get tidier and more organized than this, and vvis couldn't be happier.

Some of you might be thinking that this is too much work to create many horizontal hints across all the open space in the map and then align them together, and…slow down, no need to panic! You do not have to create many brushes, just one big brush per height; do not worry if your brush is huge and goes through world geometry as this is perfectly normal. To ease your mind, check the following screenshot showcasing the above hints in 2D.



The horizontal hint brush is 5879 x 6228 units and it goes through all the map geometry and practically covers the whole map.

If some of you are still uncertain about these horizontal hints, then you need to pay attention to the following example.



This is just a small test map featuring a central brush (the wall in the middle) that separates the 2 areas along with some props (rail, arches, and trees). The engine with all this data available will usually divide the visleaves as shown in the pic: 1 visleaf that I labeled A on the left and another one on the right labeled B (and one above the wall that I didn't show here for the sake of clarity).

If you are the player and you are standing in "A" right next to the arch, you shouldn't be seeing any of the props that are in "B" because of the wall in the middle that blocks your view. Now if you go back to paragraph II, you will notice that we said the BSP uses "visibility from a region" approach.

To refresh your mind, I said exactly this:

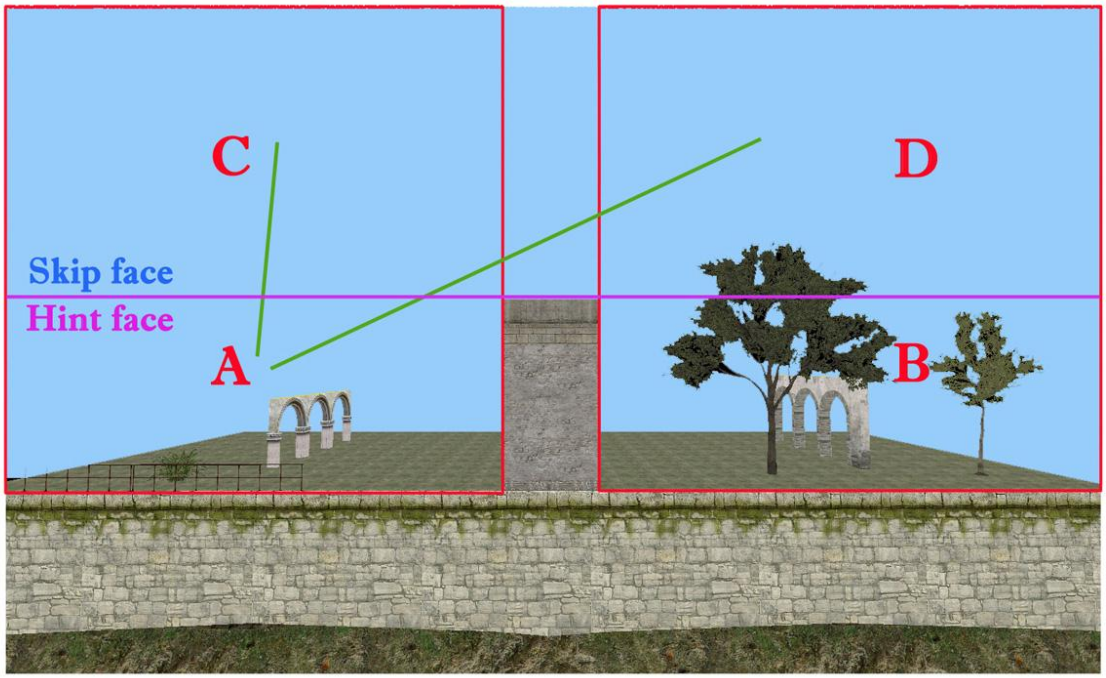
"In simpler words, the visibility from a region is pre-computed during compile time and depends on the visleaf that the camera is in (the player is actually a camera) and not on the actual location where the player is standing. If you are standing in visleaf A and the pre-computed data in the PVS instructs the engine that visleaves B,C and D are visible, then the engine will draw these leaves and their content regardless of where you stand in the visleaf A, and even if you, the player, cannot directly see these visleaves."

What does that translate to in our example? Without further information or world brushes, the engine will create a visleaf that goes from ground level to skybox level, which is the case of A and B in the pic. I have drawn a small green line between them to showcase that they

can “see” each other (they have a direct LOS). “A” and “B” are in their respective PVSs and one will be visible whenever the player is in the other one. This simply means that whenever you are in A, visleaf B’s contents will ALL be rendered, regardless of where you are standing in A and whether you actually see these contents or not.

Tsk, tsk! What a waste of resources. Don’t you agree? Keep in mind that this is a small map and nothing major is in stake; but what if your map is like de_inferno (open-ended with low houses). Imagine you are in T spawn and the tall visleaf you are in can see all other visleaves across the map; the engine will practically render all the content of these visleaves and you will have a superb slide show of horribly low frame rate (and possible map crash).

Let’s see the example of above but this time with our horizontal hint brush in place.



The only thing that changed is the addition of the horizontal hint brush that is neatly lying on the middle wall’s top (hint face down, skip face up). The engine (vbsp more specifically) has now additional information to divide the A & B visleaves; we now have 4 visleaves: A, B, C & D. You can immediately see that A has direct LOS with C and D but not with B anymore (if you have to cross the skybox or world geometry, then it is not direct LOS anymore).

Now if you are standing in A, visleaf B is not in the PVS anymore and none of its content is rendered. C & D are visible from A but they are mostly empty (or have very low content), therefore, they won’t affect your frame rate despite the fact that you can see them from A. With

this scenario, your fps will improve hugely and you will notice that the gameplay is more fluid due to fewer resources that the engine has to handle.

If you understand this concept, then you basically understood hint brushes and how to efficiently use them to cut visibility. And to re-quote myself from paragraph II:

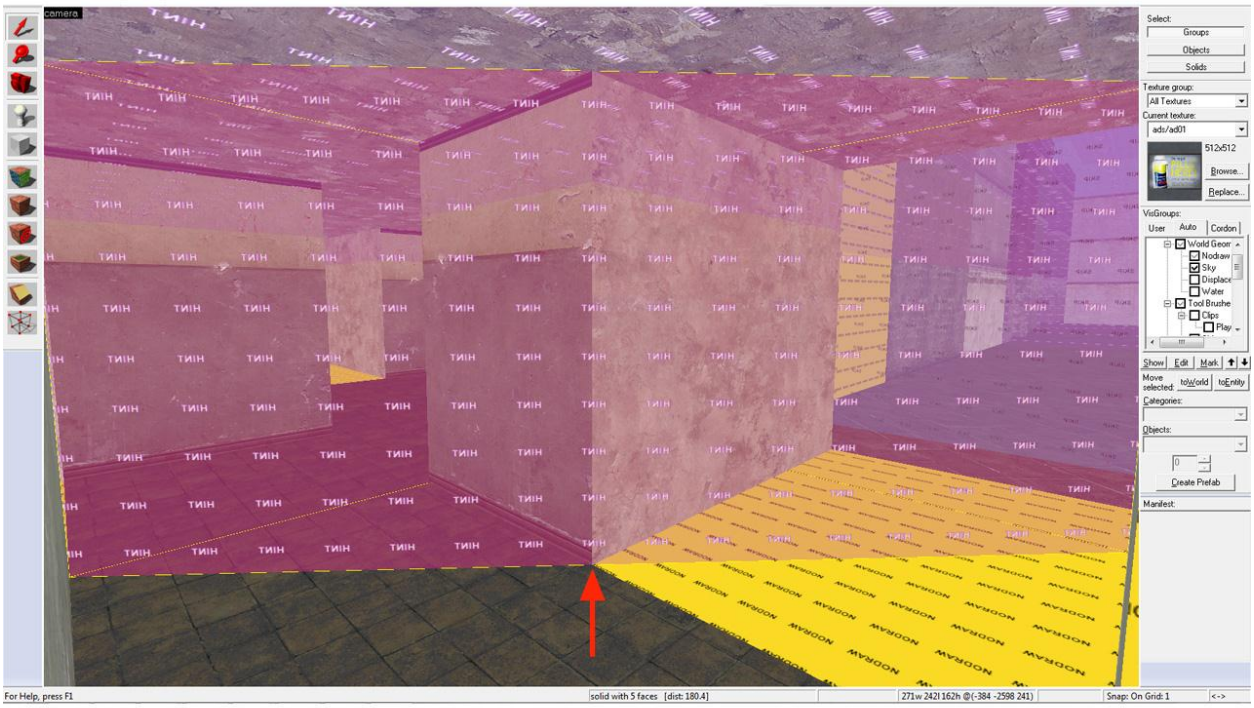
"Your ultimate goal is to make a specific visleaf "see" the <u>least</u> amount of adjacent leaves thus preventing the engine from rendering the content of these "unseen" leaves which will reduce engine overhead and increase frame rate."

This should be your guide for the correct placement of hint brushes in your map.

Another candidate for hint placement is whenever you have a window, door, doorway, hallway end. It is usually a good idea to place a hint brush in these locations to avoid having long visleaves that poke from one side to the other. Keeping long visleaves means a higher probability that this visleaf will see multiple adjacent visleaves and will get their content rendered.

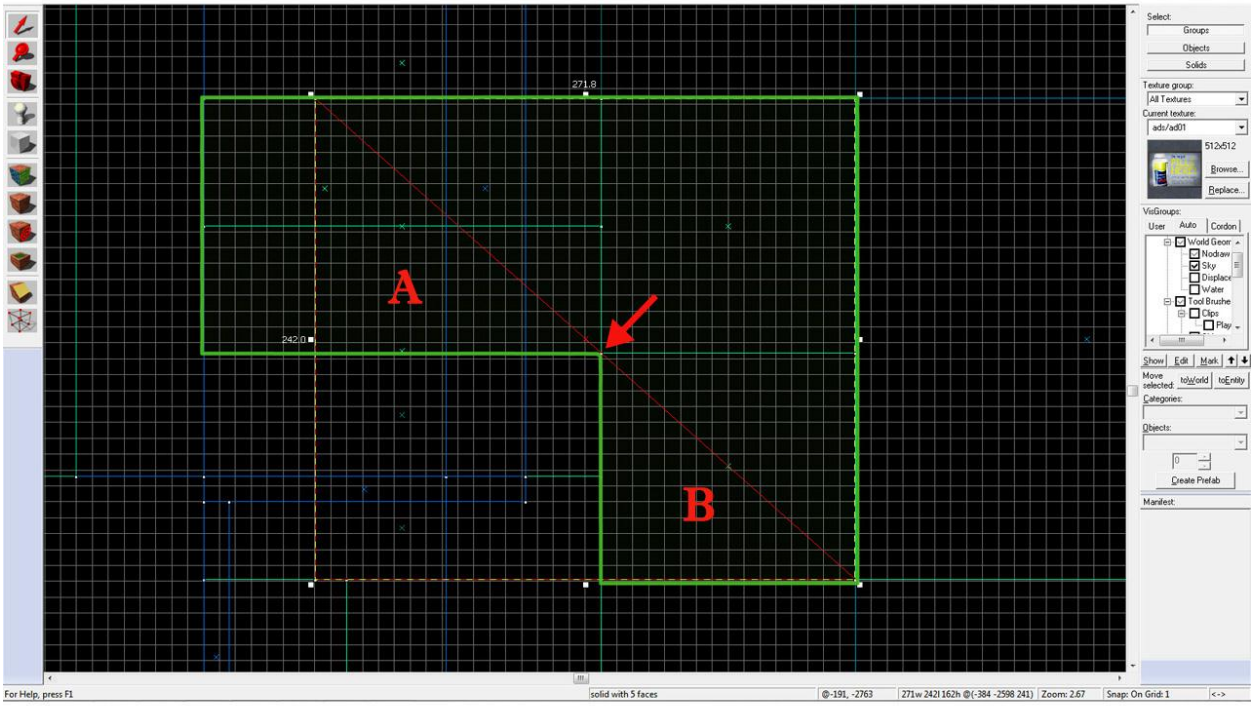
Hints are not always rectangular brushes; they can also be triangular to be able to have diagonal visleaf cuts (the shape can be anything as long as the hint face is properly aligned, but I find that a triangle is easier to manipulate in Hammer).

Remember in paragraph III.1 we discussed the layout planning and how corners should have these angular hints; this is the time to show you how to place these brushes that are almost always forgotten/overlooked by designers, even experienced ones.



This is a typical indoor corner from de_spezia_pro. Notice how the face with hint texture is perfectly touching the tip of the corner (red arrow location); all the other faces of this triangle are textured with “skip” (unless you want another face to also cut visleaves, then you should texture it with hint).

To make it easier for you to see this brush, let’s switch to 2D view.

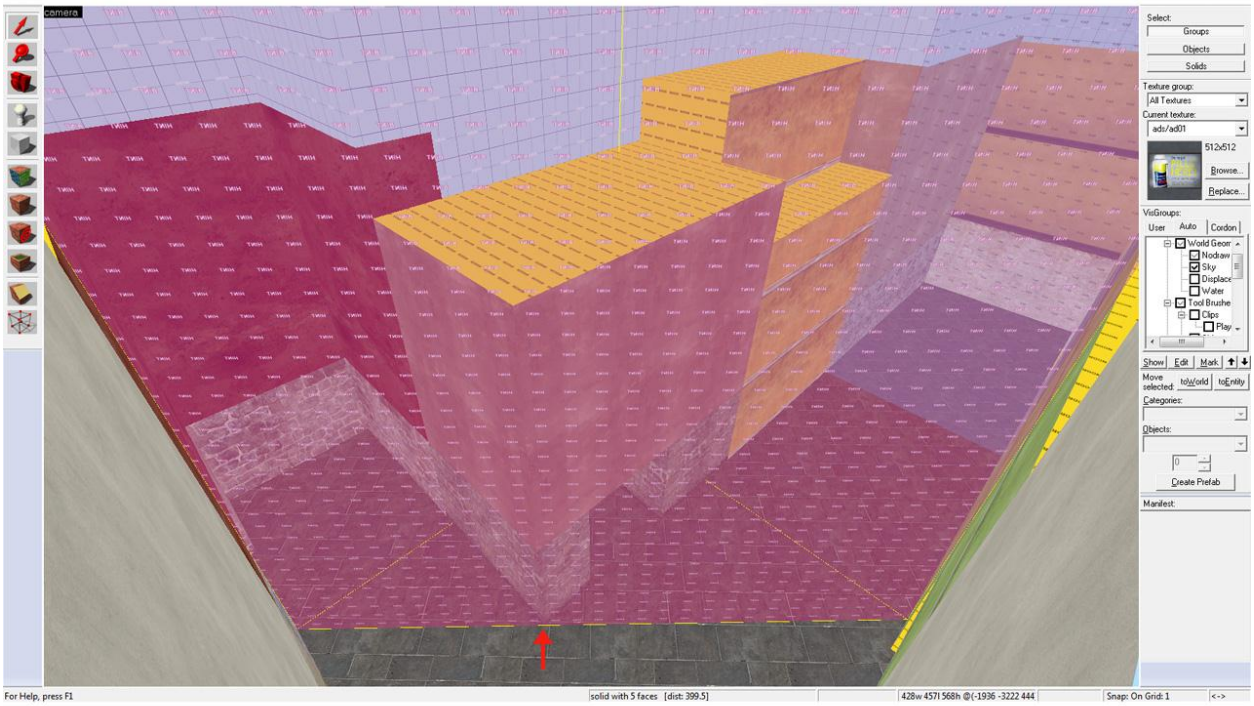


This is the same brush from the 3D view, now viewed in 2D. The red triangle is the hint brush with the hypotenuse (the diagonal side) textured in hint while all other sides are textured with skip. The corner tip is marked with the red arrow and the walkable area is highlighted in green. It is ok if the skip faces go through walls into non-playable areas since these faces will be deleted in-game. With this angular hint placement, the visleaves A & B that are on each side of the corner will not have any more direct LOS.

The most common mistake that I see mappers do is ending the hint brush abruptly at the corner tip without extending it to the other opposite wall. This won't give you the benefits that you will get from the setup I made in the pic above. Visleaf B will be extending well behind the red arrow, and not only will the contents of A be rendered, but also whatever visleaves that are beyond A to the left. Always make sure that your hint is extending on both sides of the corner until it touches a world brush.

Quick tip: sometimes you might have trouble aligning the hint face with the corner tip if your brushwork is not of standard size. In this case, you can temporarily force the hint brush to be off-grid by ALT+left-dragging the brush edges until the diagonal face aligns with the corner tip at the red arrow location (just make sure that the edges of the diagonal side keep touching the walls).

Outdoors are no different if you recall me telling you that outdoors are concealed indoors. There are some large open streets in de_spezia_pro, but they are just large scale corners/corridors if you look closely and the same angular hint can be applied.



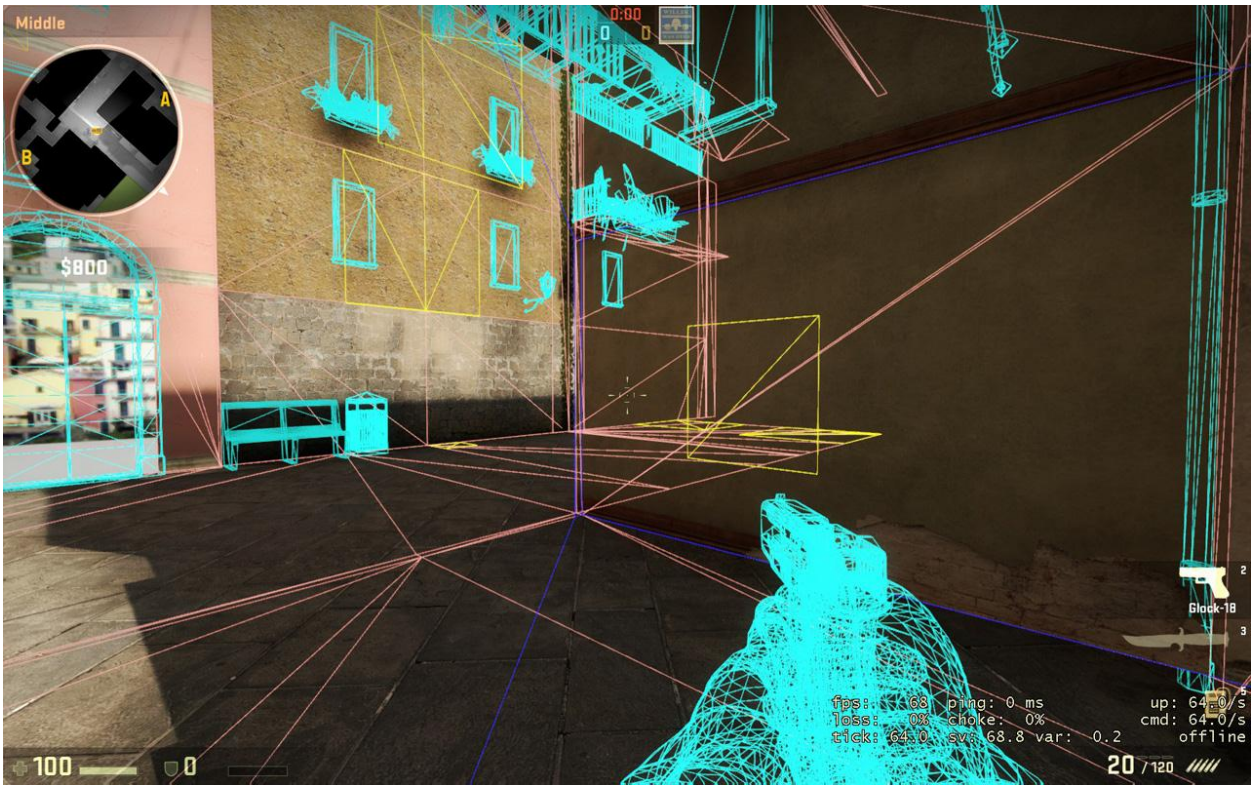
You can see in the above screenshot the triangular hint brush with the hint surface facing towards us while the other faces are "skipped". The red arrow points to the corner tip where the hint touches the corner edge. This setup will ensure that visleaves on both sides of this corner won't see each other. The player that is on one side won't have any content rendered from the other side, and this can only mean one thing: higher fps.

To see hints in action and rendered visleaves in-game, open your game console, enable cheats (sv_cheats 1) then type mat_leafvis 1 (to see individual visleaves) or mat_leafvis 3 (to check the PVS or how many visleaves the visleaf you are standing in is seeing). You may also want to enable wireframe mode by typing mat_wireframe 1 in console, to see what the engine is drawing in real time. This will help you decide whether your hint brushes are doing their job in cutting visibility or you need to alter some locations and add some others.

This screenshot is from the main street in de_spezia_pro.

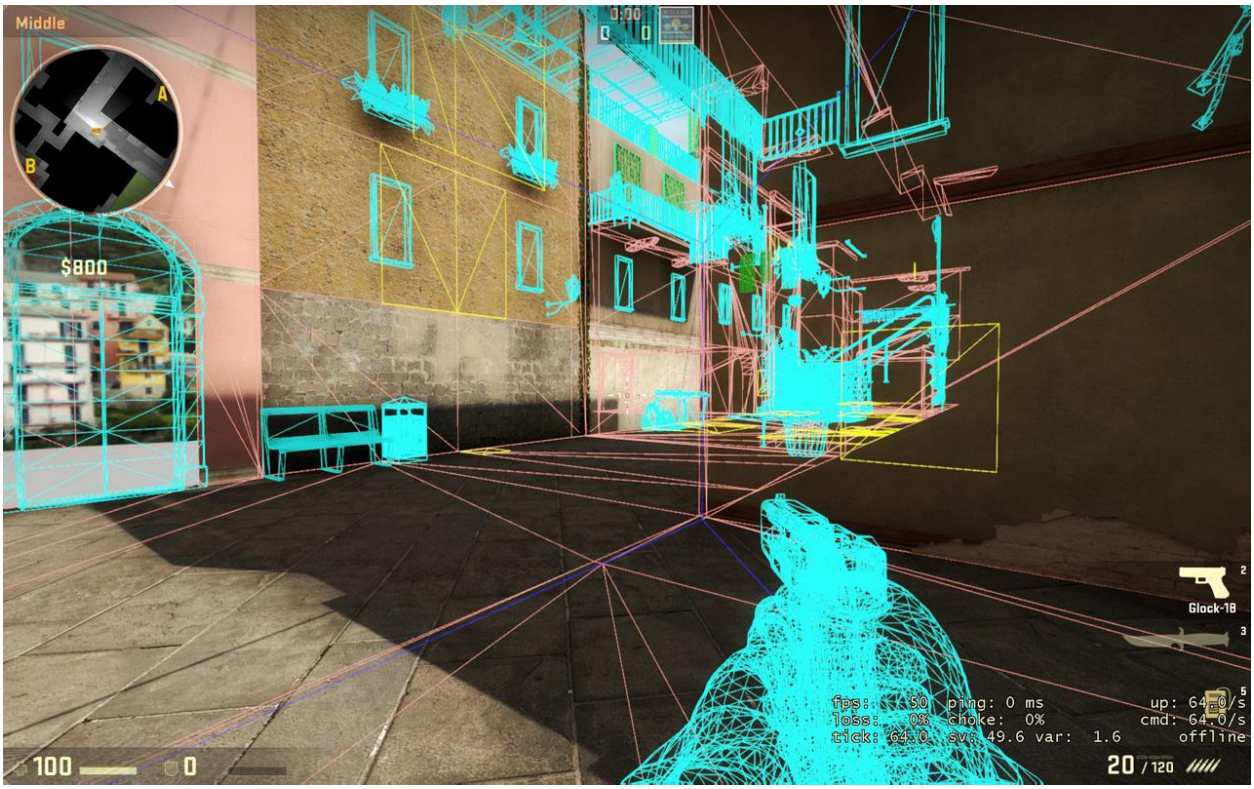
The red arrows indicate the diagonal visleaf cut as a result of the angular hint brush that I placed on this corner.

To show the effect of these hints in defining visleaves and cutting visibility, check the next 2 screenshots (with mat_leafvis 1 and mat_wireframe 1).



Here I am standing to the right of the hint diagonal cut (keep in mind the earlier 2D view screenshot showcasing A & B visleaves on each side). You can obviously see that the street on the opposite side of the corner is not drawn.

If I move couple of steps to my left past this diagonal line, this engine will render this.



You can see that most of the street is rendered now and that's normal because I moved to a visleaf beyond the hint brush location and this visleaf can see the other visleaves on the other side of the street.

Without the hint placed on the corner, the engine will always draw the full street content, wasting a lot of resources along the way and automatically lowering your in-game fps. You can clearly notice in the previous pic that the hint actually prevented the engine from drawing any content from the other street, keeping your fps as high as possible in the process.

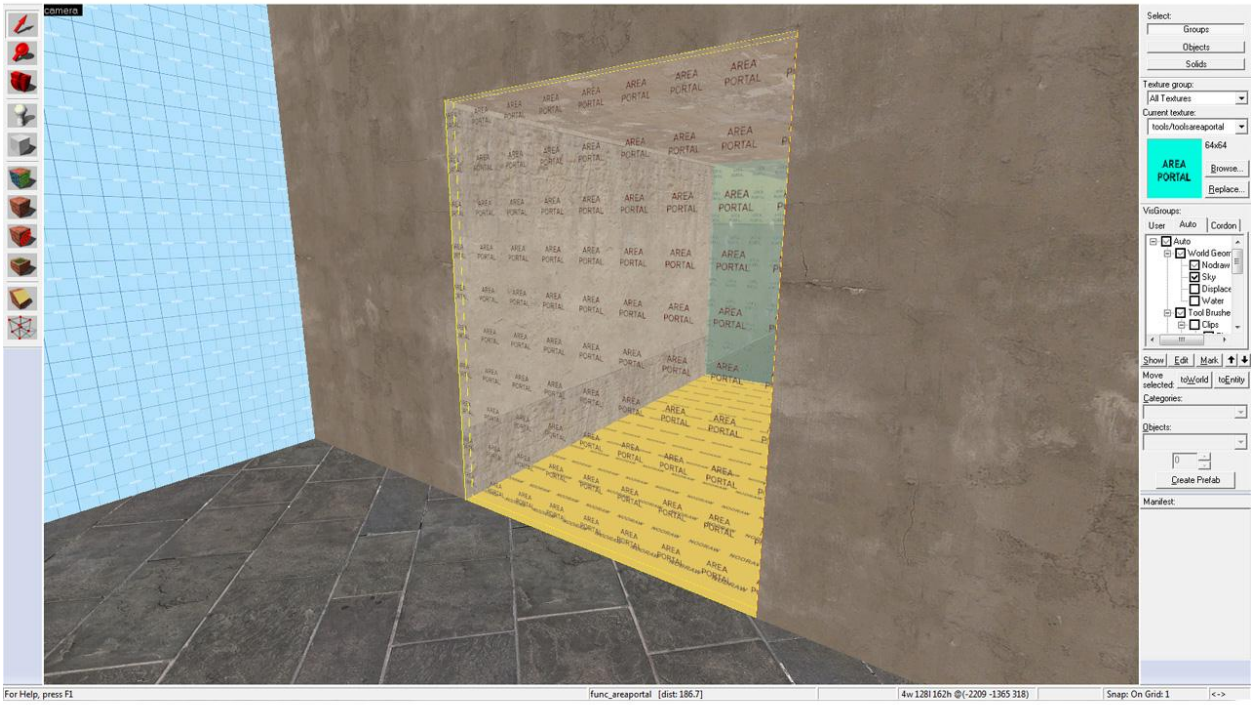
III.8 – Areaportals

Sometimes hint brushes are not enough to control visibility and that's where areaportals come in handy. If hints are your "MP5" submachine gun, then areaportals are your "M4" assault rifle ☺.

Unlike hint brushes, areaportals are brush entities that have the tools/toolsareaportal texture on all faces. Areaportals brushes are better kept thin (1 to 8 units) since the areaportal brush will create its own visleaf. In addition, areaportals will hide both world geometry and models, which makes them a truly useful tool in your optimization arsenal.

And just like hints, you need to hide all visgroups in Hammer except for the skybox and regular world geometry (and the areaportal group obviously); this is to make sure that your areaportals will be correctly placed between brushes that are actually taken into consideration for

visibility and visleaves calculation. In the case of areaportals, it is more important than is the case in hints to make sure that the func_areaportals are touching world brushes or skybox brushes with no gap what so ever; otherwise you will have a leak during compile just like a normal leak.



This pic is from de_spezia_pro near bombsite A. Notice how I only kept the skybox and the square world brushes to be able to clearly place the areaportal exactly between those brushes at the hallway end. Areaportals are usually placed on doors and windows, but can exist anywhere in the map. Typically, I use them at the end of hallways, corridors and even in open areas for separating streets.

Areaportals can be of 2 types: func_areaportal and func_areaportalwindow, with the first one being used more extensively across maps.

For func_areaportal, you can control its state, to be open or closed. This is done by accessing the areaportal's properties window and setting the initial state to either open or closed.

To check areaportals in-game, open the console, enable cheats (sv_cheats 1) then type r_drawportals 1. All areaportals will be highlighted in green; open areaportals will have double edges (outlines) while closed ones will have single edge.

Each green rectangle represents an areaportal shown at the exact location where you placed it in Hammer. Areaportals that are on the same plane (such as 2 areaportals on 2 adjacent windows) will be grouped together in-game.

The above pic shows a closed areaportal with a single line green edge (see red arrow)

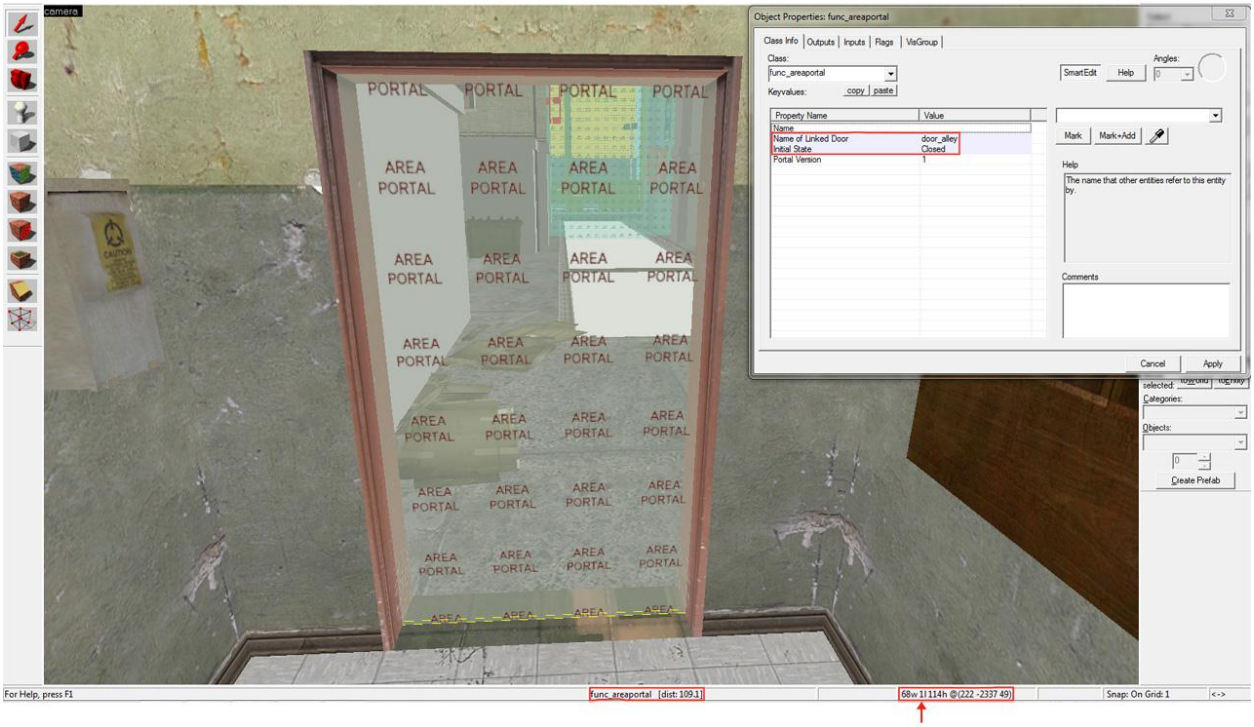
In this pic, you can see an open areaportal denoted by the double green line edge (red arrows location).

Now that you know the difference between open and closed areaportals, let us go into details about them.

Closed areaportals have their initial state set to closed, and they are linked to a closed door. The screenshot below should make it easier to understand.

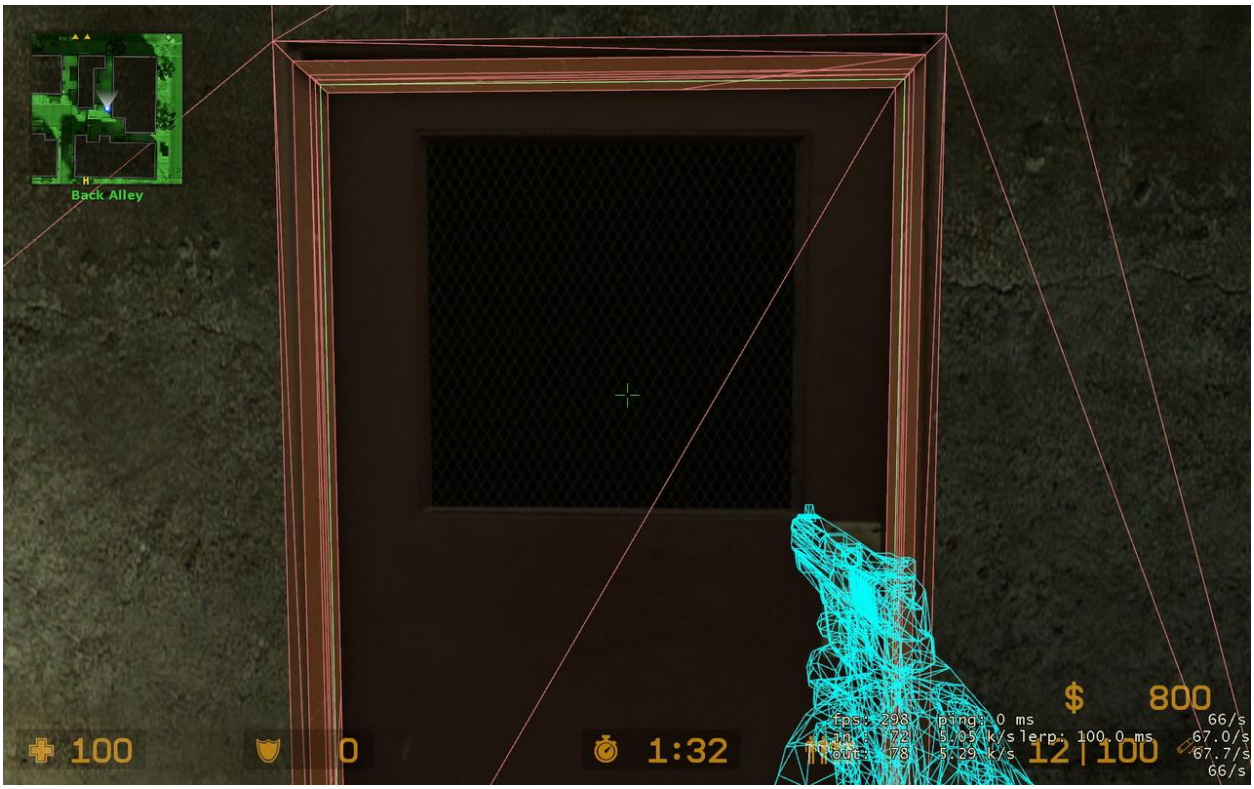
You need to create the door first; in the pic from cs_east_borough, I made the door as func_door_rotating (brush) but feel free to make it out of prop_door_rotating (model). The door is named “door_alley” (left arrow location) and the name is important since it will be used to link the areaportal to the door. Notice that the door has a thickness of 3 units (right arrow location). It is crucial to keep this number in mind because you need to make the areaportal thinner to be able to insert it inside the door completely. If the areaportal is poking through the door, then the whole process is screwed and you will have visual glitches (void rendering).

Let’s hide the door to see the areaportal inside.



If you check the red arrow on the screenshot, it shows you that the areaportal thickness is 1 unit, which is perfect to completely hide it inside the door. In the properties window, you simply input the name of the door in the “name of linked door” field and you set the initial state to “closed”. Please note that if your door starts open in the map, set the initial state to “open”; the areaportal state simply has to match the door’s initial state.

We know how to setup the closed areaportal, how about we see it live in action.



Always remember that you need mat_wireframe 1 to test visibility in-game.

This is the closed areaportal inside the door: If you take a closer look, you will actually see…nothing! Great, isn't it? The closed areaportal is preventing the engine from rendering anything behind it, how cool is that.

If we open the door, the areaportal will automatically switch to open state. The engine will render the content of visleaves that are behind the door/areaportal until you close the door back or the door automatically closes. As soon as the door closes, the areaportal will close and the engine will once again stop rendering anything behind the door.

You might be wondering: what if I close the door and another player is behind it, will he see the void? Of course not. You do not have to worry about this issue since areaportals work on the client side and each player will see the world based on what areaportals are open/closed on his side.

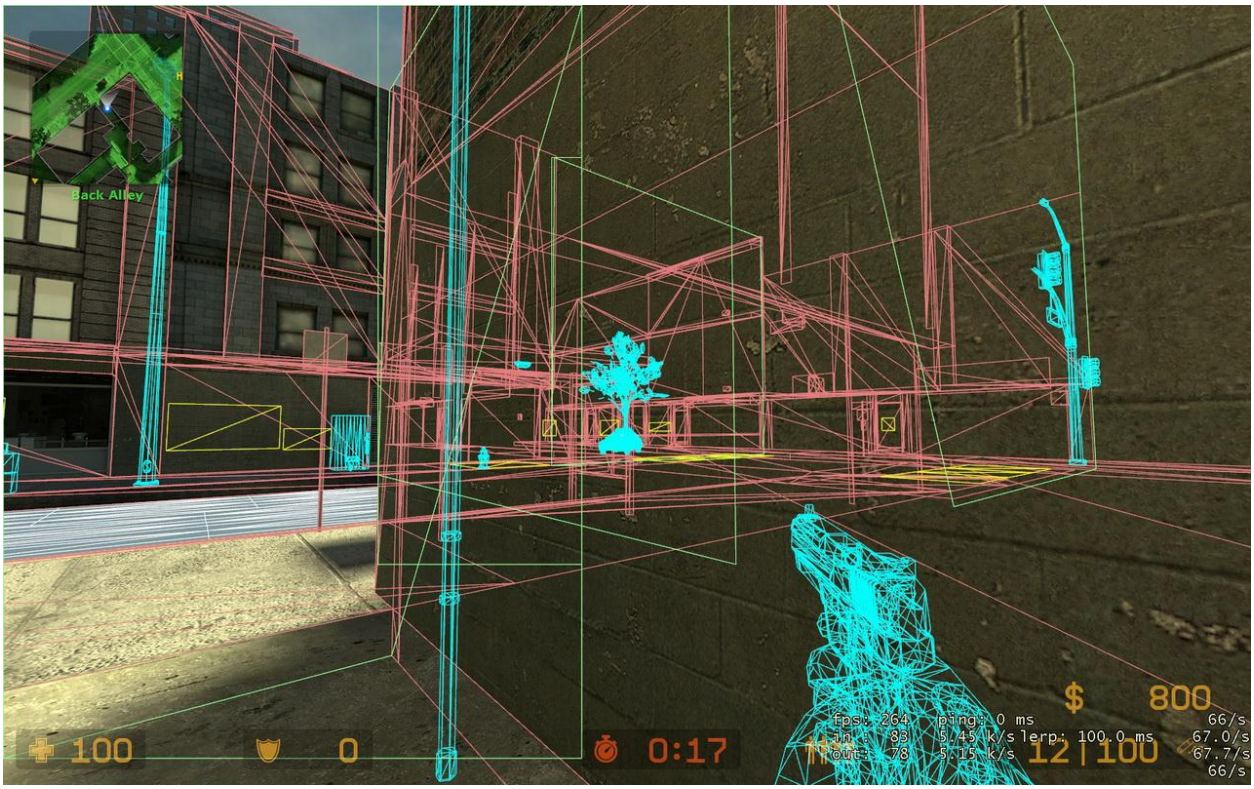
This is the open areaportal once we opened the door. You can see that the engine is rendering the content of the visleaf behind the door since the areaportal is open now.

Not all maps have doors that open and close, and as I said earlier in this paragraph, areaportals can be used anywhere especially on windows, doorway, hallway ends and open streets. Obviously in these cases, you can't set the areaportal state to "closed" as the player will see the void behind it which is totally unrealistic and level-breaking. So are we stuck? Not quite.

It's true that for an areaportal located in the middle of the street, we cannot set the state to closed. Let us set it to open then. But the engine will draw what is behind the areaportal, I hear you say.

Nothing is lost when you set the state to "open". Despite not hiding the world behind it, the open areaportal will present a very nice effect called culling or more precisely, view frustum culling. It is the process of removing objects that lie completely outside the viewing frustum, the field of view of the player. Instead of hiding everything behind it, the areaportal will selectively hide the world that you, the player, can't see from your position. Let's see an example from cs_east_borough taken from the edge of the back alley where it meets the main street.

I'm standing at the edge of the back alley (refer to radar on the top left corner of the screenshot). I am close to the wall and the street beyond is barely rendered. If I move couple of steps to the left, my view frustum will change and the areaportal will update the visibility accordingly.



You can see that a couple of props (blue) and buildings' facades (pink) are rendered now. If I keep moving to the left, the whole street will become visible to me and the areaportal will allow it to draw completely. Without the areaportal, the content of this street will be all rendered even if I have this corner edge in front of me, simply because the visleaf that I'm standing in has direct LOS to the street visleaves. Hints in this situation cannot cut visibility while areaportals can, even without cutting further visleaves (you are still moving within the same visleaf and getting visibility under control, something that the hint cannot do).

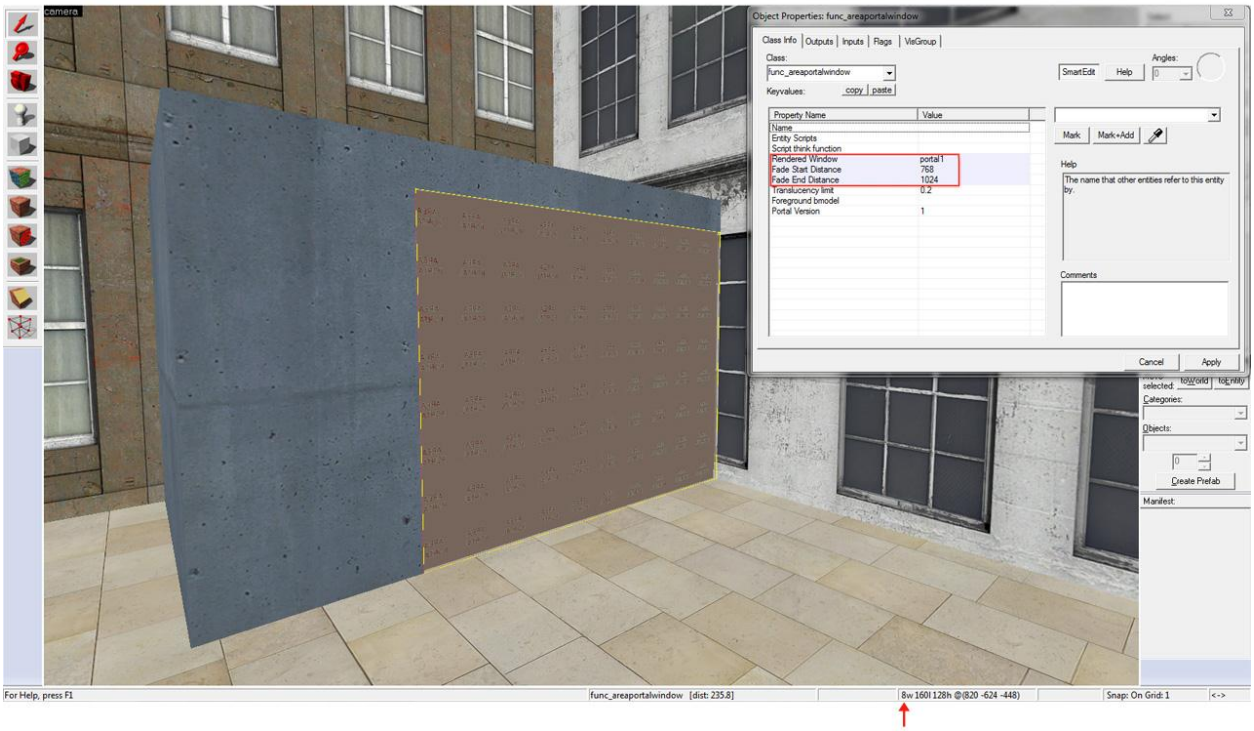
You can clearly see the tremendous benefits of areaportals in controlling visibility and forcing the engine not to render parts of your level, which will translate in higher frame rate and happier players.

As mentioned earlier, a 2nd type of areaportals, much less used, is also available; func_areaportalwindow. As the name suggests, it is mostly used on windows, but can also be used anywhere you like.

It differs from the regular areaportal by not having open/closed state, but rather a fade distance that you can control, and beyond which the areaportal will completely close and stops rendering what's behind it. A small problem arises here: when the areaportal closes from a distance, the player will be able to see the void behind it since the world will not be rendered.

To remedy this issue, it is always advised to link the areaportal to a func_brush, preferably textured in black/grey/white, to enable the engine to render this brush and hide the void once the areaportal closes.

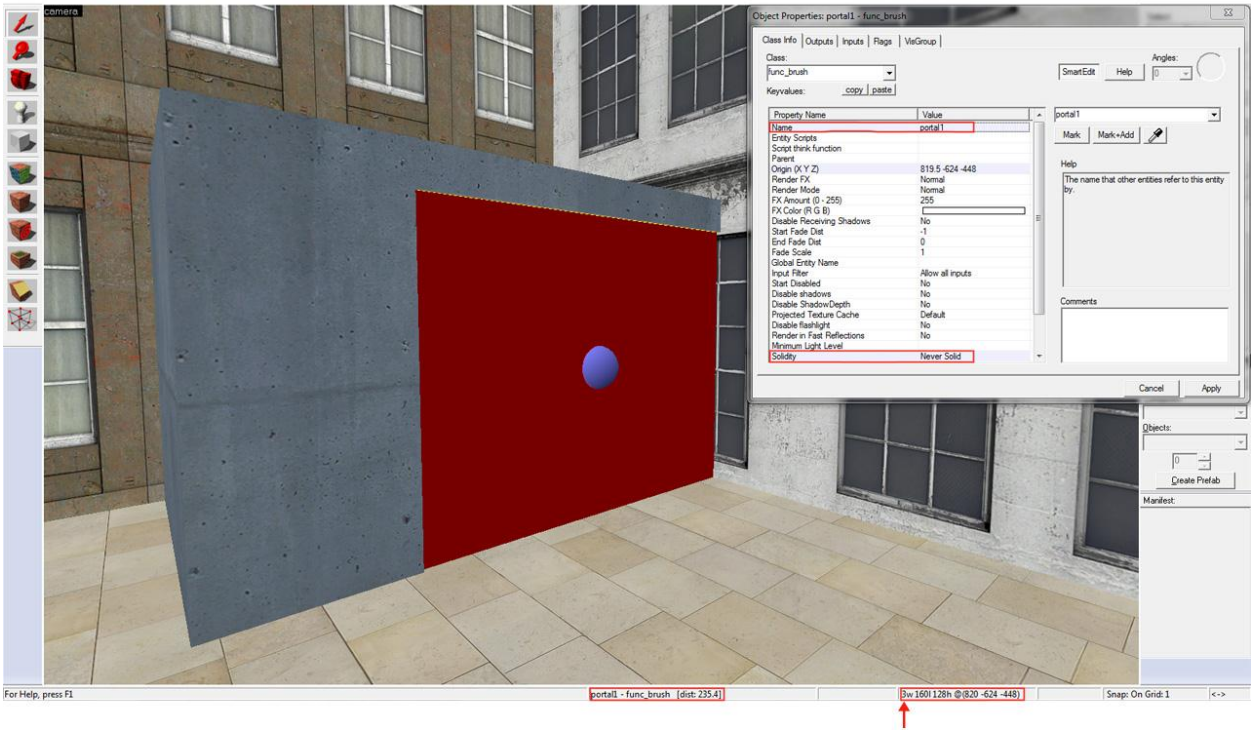
Moving on to the screenshots.



Instead of linking to a door, you will link to the opaque func_brush; simply enter the name of the brush in the “rendered window” field, then set the fade distances, just like we did with the props fading distance in paragraph III.6. The distances will depend on your map layout and surrounding geometry, just make sure not to fade it too soon as this will break the gameplay and realism in your level.

The numbers above simply mean that when the player is at a distance of 768 units from the areaportal, the fading effect will start. When he reaches 1024 units, the fade is complete; the areaportal fully closes and stops rendering the visleaves behind it, and renders the func_brush instead.

And just like the regular areaportal, it is equally important to note the thickness (8 units in the pic-red arrow location), to make sure that the linked func_brush will be thinner and will be totally enclosed inside the areaportal.



If we hide the areaportal in Hammer, we can finally see the linked brush. It must have a name, to be used to link it to the areaportal. The width is 3 units (red arrow location), just perfect to be completely enveloped inside the areaportal (8 units). The solidity should be set to “never solid” otherwise players won’t be able to cross it, and the brush should be textured in black or white/grey-ish.

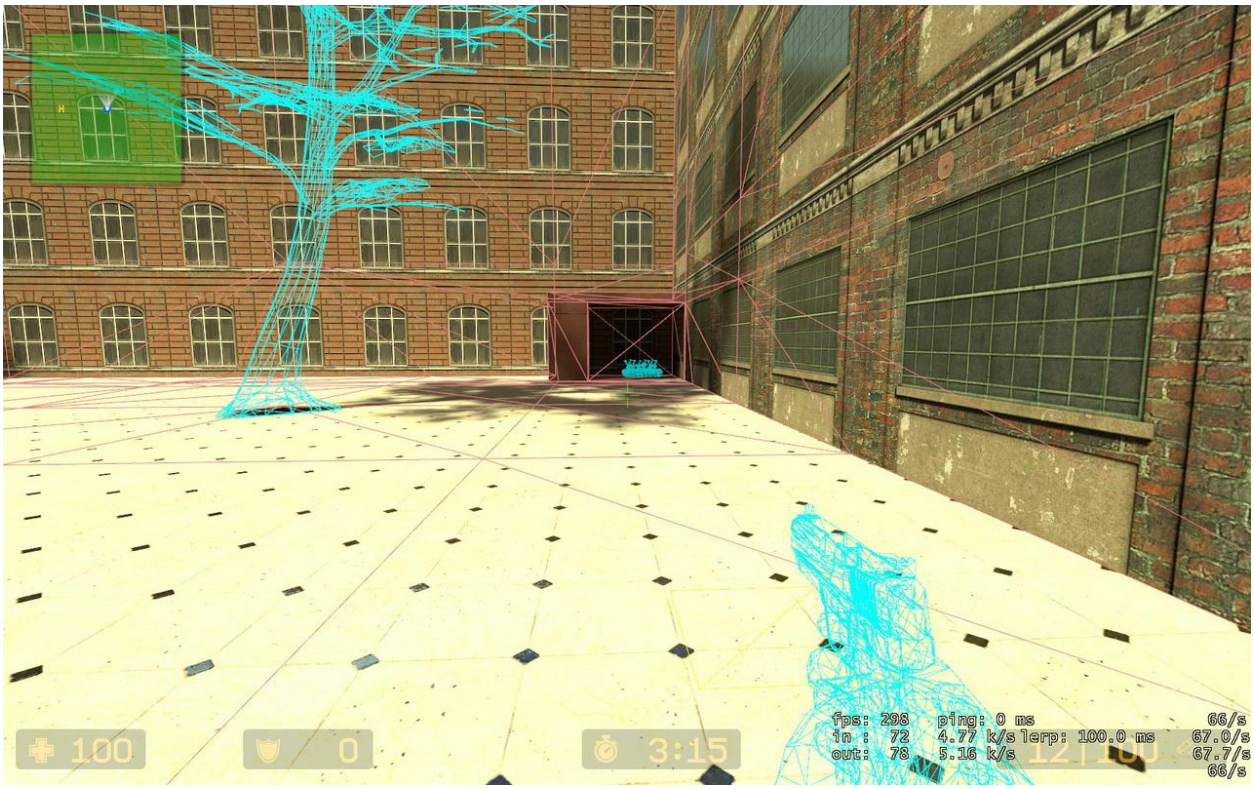
To see this in action, take a look at the following screenshots.

I am standing here beyond the end fade distance (1024 units in our example); the areaportal is fully closed and the black func_brush is rendered instead.

Here’s the same pic in wireframe mode. Notice that the inside of this small room is not rendered due to the areaportal being closed at this distance.

If we move forward, the brush starts fading out until we reach the distance of 768 units from the areaportal; at this distance, the areaportal is fully open and the inside visleaf is fully rendered.

To see it clearer with wireframe, check the next pic.



You can already see the window and the flower bed inside, and the black brush is not rendered anymore. Also note that the view frustum culling effect applies in the areaportalwindow too; if you start moving sideways, the areaportal will start culling/removing content that is not in your direct field of vision.

One final word of advice though about "func_areaportalwindow"s; they can create unfair advantage and unbalanced gameplay, especially in multiplayer games. This is why they are less common than their regular counterpart, the func_areaportal.

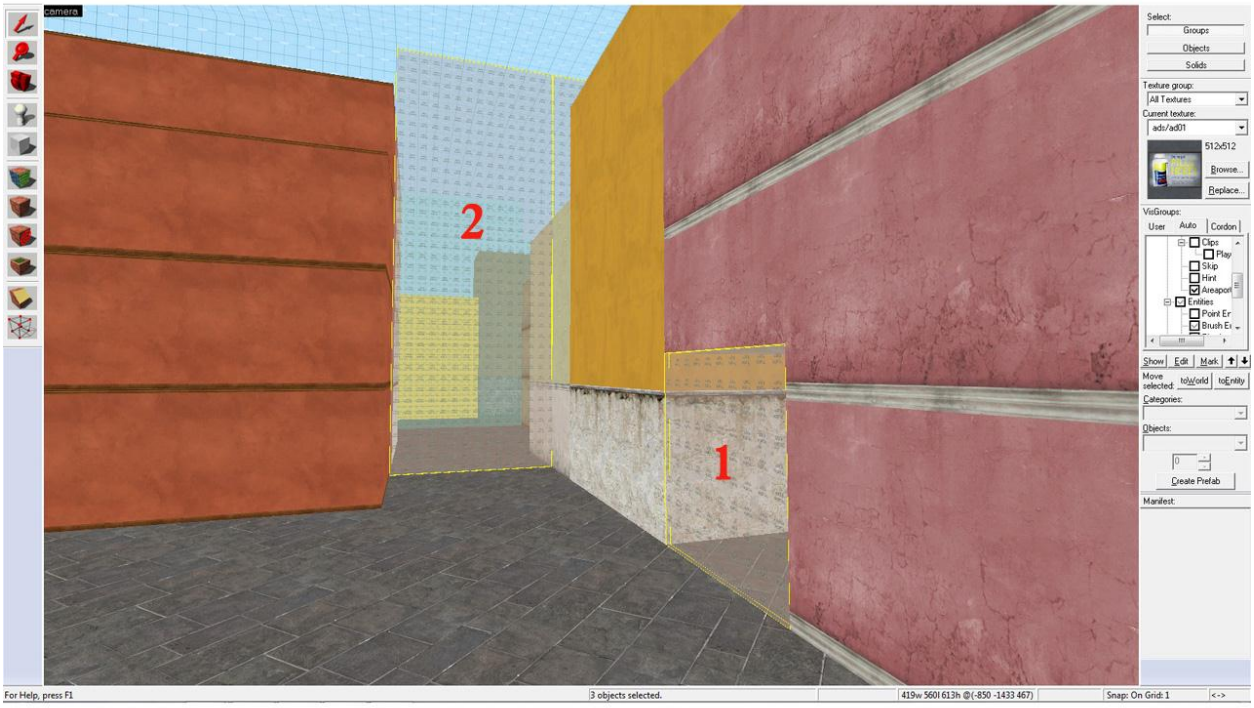
Consider the above screenshot taken in Counter-strike Source, a competitive multiplayer game. If I'm charging towards this small room and I'm still at 1024 units from it, then nothing inside is rendered and all I see is the black brush that blocks the view. If another player is defending the room, and he is sitting at 32 units behind the areaportal, then the areaportal will be open for him and he can clearly see anyone charging towards his position.

You can easily see the unfair advantage in this situation as he's perfectly capable of seeing and knocking me down, while I have no clue of his whereabouts until I'm well inside the 768 units distance. Being in a competitive realistic game, the defending player would have already shot me long before I reach the 768 mark.

To remedy for this, you need to provide enough cover for the attacking team and/or scatter some visual block for the defending team. A good example is found in the Counter-Strike Global Offensive map de_safehouse, where attackers have enough cars, vans, rocks to take cover

and advance while defenders inside the house have some fences and trees to partially block their field of view and give the attackers a fair chance of advancing. If you're not feeling comfortable about finding the right balance with areaportalwindow, then you can always switch back to using a regular areaportal.

And finally, check this screenshot to show you that areaportals are not only restricted to corridors, hallway ends and windows/doors, but they can also be used to separate large streets into different areas as I already mentioned in this paragraph's beginning.



This is de_spezia_pro at CT spawn. Labeled 1 is the small size areaportal that is traditionally placed at the hallway's end to separate indoors from outdoors. Labeled 2 is a rather tall and large areaportal that goes from ground level to skybox and that is used to separate CT spawn area from the middle street. (If you look closely behind areaportal 2, you would notice a third tall areaportal at the other end of the street; always get in the habit of tightly closing your areas with areaportals).

And remember, to see the effect of areaportals on visibility, type mat_wireframe 1 and wander around near them in-game. You will start noticing parts of the world appearing/disappearing as you move around and that's good news; your areaportals are working and you did fine in placing them.

III.9 – Occluders

If you followed my systematic approach for your optimization process, then by now, your map should be nearly bullet-proof when it comes to frame rate. However, some maps might still need an extra step to be fully optimized. The last resort in optimization is by using occluders.

Like areaportals, they are brush entities that have the "tools/toolsoccluder" texture on the occluding face and the nodraw (or skip) texture on all other faces. Occluders are used to hide models only and not world geometry. Unlike areaportals, occluders do not need to seal an area completely; they can exist anywhere, inside a prop or a brush, touching another brush or floating mid-air.

You need to keep one thing in mind when working with occluders: they are very costly for the engine as they work in real-time to decide which models get rendered and which don't, based on your relative location to the occluder brush. You might be asking why occluders are costly; simply because their visibility calculations are done in real time while you are moving around in the map, and not during the bsp creation like hints and areaportals.

If you go back to paragraph II, I briefly discussed the issue of "visibility from a point" approach (used by other games which is mostly implemented in real time, and visibility is determined from the current viewpoint only) versus "visibility from a region" approach that the BSP uses. For hints and areaportals, all the visibility calculations and numbers' crunching occur at the vvis phase of the compilation process (that's why your CPU usage peaks to 100% usage during compile). All the results are stored in the BSP tree for easy access later during gameplay with minimal engine overhead or resources usage.

In the case of occluders, nothing is pre-calculated or stored in the bsp and the engine has to work out these calculations on the fly while you are moving in the map, depending on your actual viewpoint or view frustum relative to the occluding face. Every time you make a step to the left or right, jump, duck or look the other way, the engine has to re-calculate the visibility matrix and decide which models to draw and which ones to hide based on your updated location in the map, and all this has to happen in less than a millisecond (otherwise you'd have a slideshow on screen).

That would be ok if the engine only has the occluders' visibility to process, but as you know, this is not the case. The engine has tons of other operations/resources to calculate and take care of using your available CPU/GPU/RAM power and capacity. You can easily deduce that giving the engine a lot of occluders to calculate their visibility is not a good idea as other calculations will suffer and your fps will inevitably decrease.

That is also why you should not texture the whole occluder brush with the occluder texture; only the occluding face should get this texture while other faces should get the nodraw (or skip), to be discarded by the engine in-game. If you texture the whole brush with occluder

textures, then the visibility calculations will increase exponentially since the engine has to calculate the visibility of 1 face relatively to the other faces, in addition to the visibility of this face to nearby models. Complicated, isn't it! If you think so, then think of the poor engine that has to do all these calculations while you are running near the occluder brush ☺.

I personally do not use occluders since I manage my optimization using all the previously discussed methods (but I will use them in the near future on an open-ended CSGO map that I started working on); however, hints and areaportals might not provide the solution in some maps (or even they might not be feasible to be implemented).

A prime candidate map for occluders is one with open space and very few world geometry structures. With the lack of sufficient regular world geometry and the lack of corridors/corners, hints and areaportals cannot be easily implemented, and might not yield noticeable benefits in frame rate increase if employed.

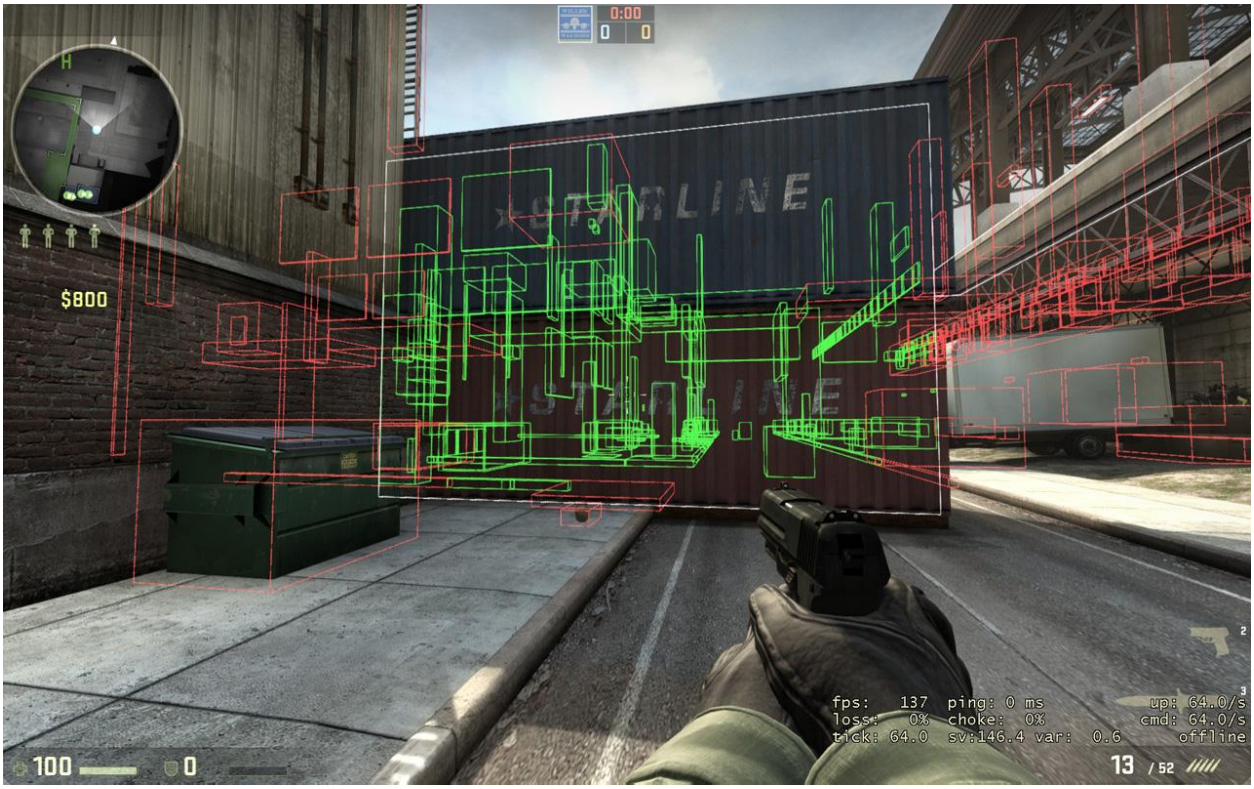
A perfect example is cs_compound from Counter-Strike Source. The map has few distant structures that are mostly func_detail and has a very open-ended layout with virtually no corridors/corners. Occluder brushes are mostly found inside the wall that circles around the compound and inside various walls of the main building and warehouse in CT spawn.

This shot is from cs_assault. I temporarily removed the upper container so you can see the occluder that is inside (both containers include occluders). Please observe the texturing: the face on our side has the occluder texture while the other faces are "skipped".

When you are standing near the dumpster, the occluding face will hide all props behind it within your viewing angle (default 90° for CS and 75° for HL2).

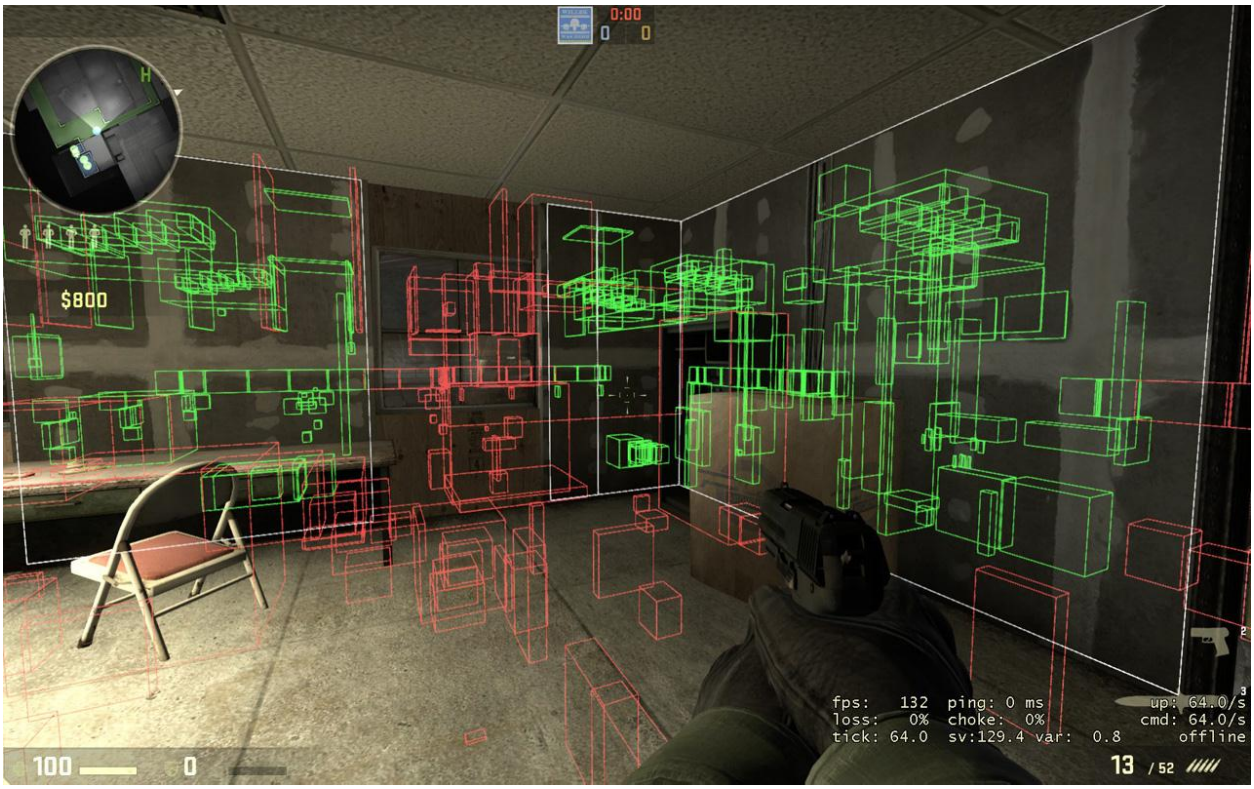
To check occluders in-game, enable cheats then type r_visocclusion 1. The occluder brushes will be outlined in white while the hidden (culled) props will be highlighted in green and the non-culled in red.

Here is an example of the above occluder in-game.



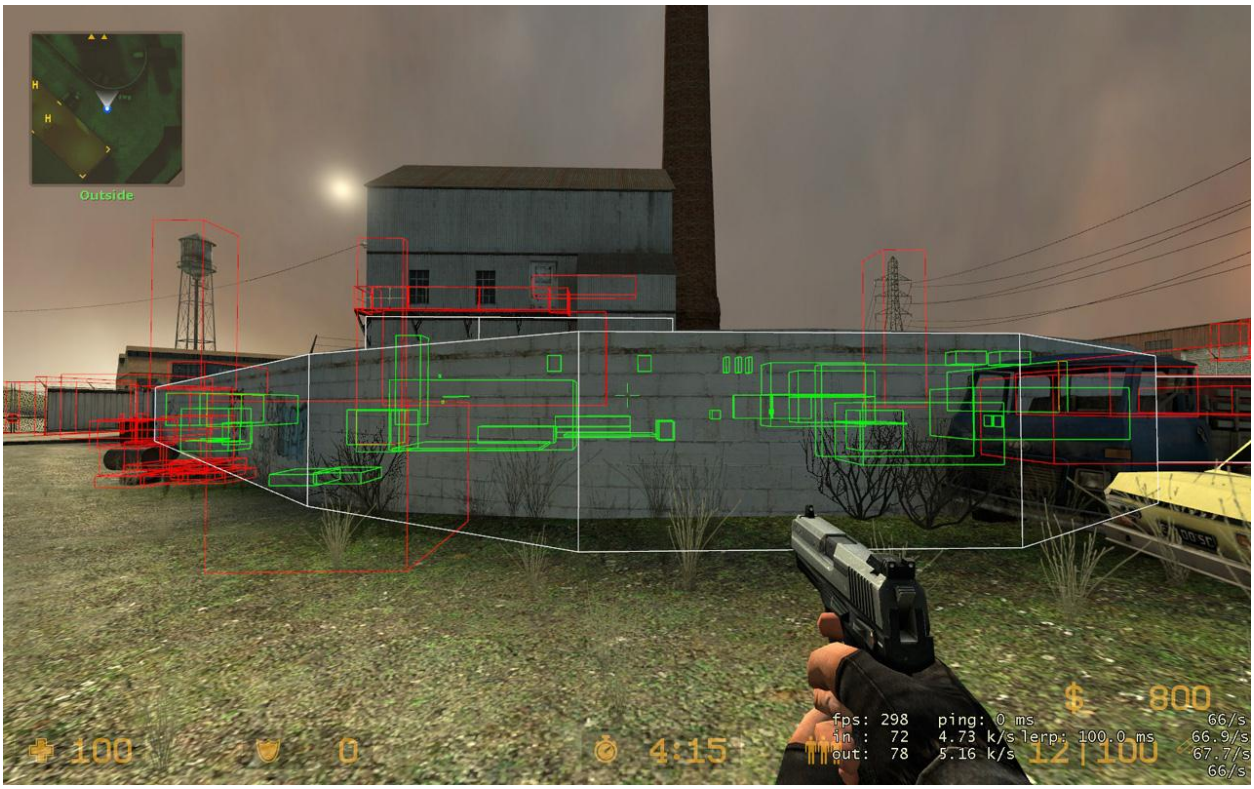
The occluder is needed here since it's an open street intersection and not much can be done with hints and areaportals to cut visibility. You can type r_occlusionspew 1 in console to get an idea of how many props are hidden on screen out of the total props available in your FOV (you will get the number and percentage of props occluded).

Another location in cs_assault containing an occluder is the room in front of the hostages' room.



The warehouse has a lot of props and you do not see them once you are at the door of the hostage room. Without the occluder (3 occluders in the pic), the engine will simply draw all of these props (well, most of them since you can reduce slightly the number with some areaportals but not easily as the area is mostly func_detail that cannot support areaportal placement). The occluding face is on our side in the screenshot; there is no need to have a 2nd occluding face on the outside since this room does not have enough props/expensive models to warrant an additional occluding face. Keep this criterion in mind when deciding which face of the brush gets the occluder texture.

Let's have another example from cs_compound.



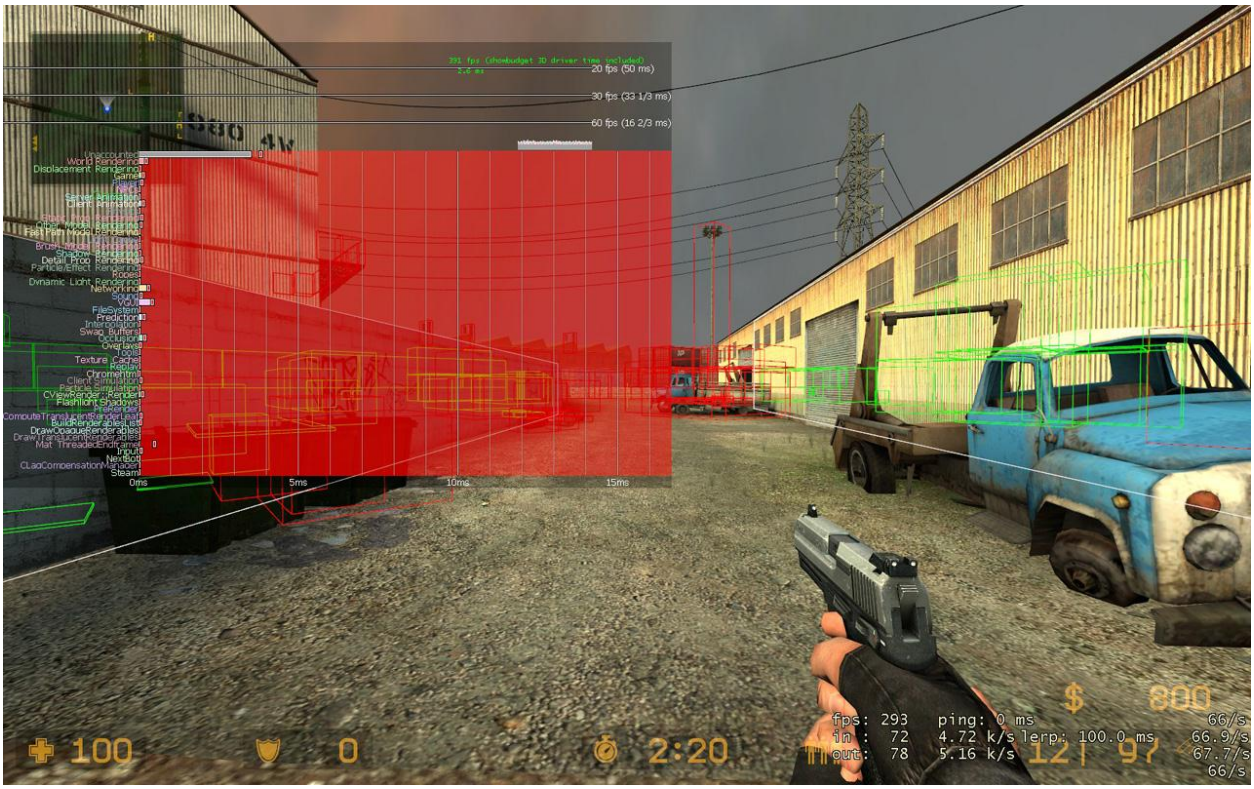
Again, several occluder brushes are well inserted inside the wall (a func_detail by the way) and they are highlighted in white. The hidden props are green and the visible ones are red.

Now all this is fine and fancy, but the question is how you can tell if your occluder is doing a good job or making things worse due to its cost.

To answer this question, you need to turn off occlusion to see the difference and judge accordingly of the occluder's necessity, or lack of.

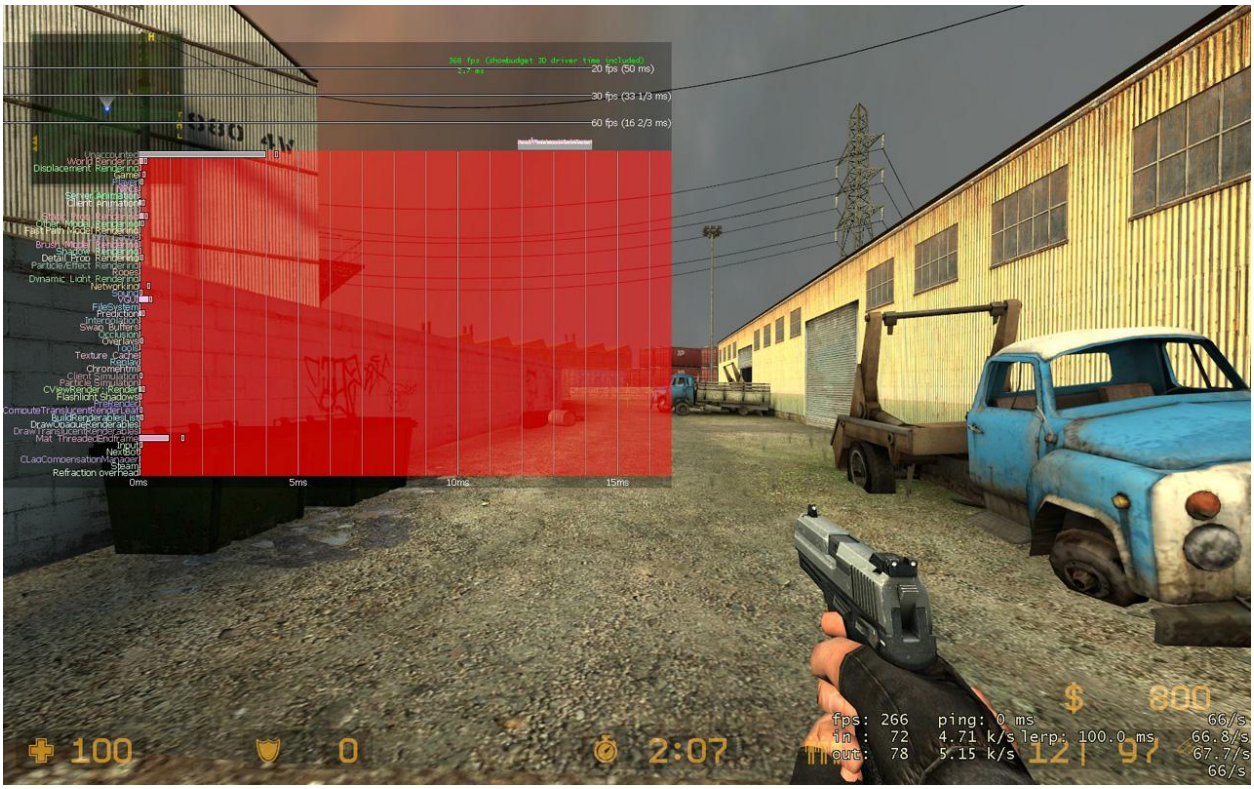
To check the effect of occluders on your frame rate, type r_occlusion 0 to turn off occlusion (sadly, Valve removed this console command in CSGO along with other useful commands). If there is no noticeable drop in your fps, then your occluder is useless and eating up unnecessary resources and is better removed. However, if after turning off occlusion you notice a severe drop in fps, then your occluder was really needed and was doing a good job cutting visibility and maintaining high frame rate.

And since we have established that you all like colorful screenshots, there you go.



This shot is taken with occlusion on (r_occlusion 1 which is the default setup in Source)

If we turn off occlusion and take another screenshot, here's what happens.



You can notice a decrease in overall fps of around 30 frames per second because, without occluders, the engine will render all the props that were hidden in the previous pic, since they are in visleaves that are seen by your visleaf. In this case, you can safely deduce that the occluder was needed here and was giving you a boost of 30 fps, totally justifying its cost.

As I mentioned earlier in this paragraph, occluders are costly for the engine. Keep trying with other optimization techniques over and over again until you are sick and exhausted and can't improve the frame rate anymore; then, and only then, use occluders.

III.10 – Gameplay & Lighting

By now, your map is perfectly optimized if you have followed this systematic approach carefully. This paragraph aims to make the map even more professional and enjoyable.

The map's gameplay will depend mainly on its layout as we discussed in paragraph III.1; if the layout is well thought and laid, then the gameplay should be fun and balanced.

But even with a solid layout, the gameplay can still be hindered by some small annoyances, in the form of players getting stuck in corners, walls and various geometric shapes in your level.

I personally hate being stuck in a map with passion. I prefer my motion to be as smooth as possible; there is nothing more irritating and frustrating than getting stuck on a small window sill or door frame, or even a bucket on the floor during an intense firefight.

What can be done then to fight this issue?

Allow me to present to you, the clip brush, which is a regular world brush that has the "tools/toolsclip" texture or the "tools/toolsplayerclip" on all its faces. The clip brush clips both players and NPCs (non-playable characters) while the player clip brush clips only players. They both allow bullets, grenades and physics objects to go through them. Clip is used here to denote the fact of limiting a player's movement and preventing him from going through the brush. The clip brush is mostly used on the map's edges to prevent players from going outside the playable area, and to cover pointy and complex-shaped brushwork, thus preventing players from colliding with them and getting stuck.

The clip brush can be used to cover another brush (regular world brush or func_detail), or a prop (mainly prop_static). Props have an additional option to further improve gameplay flow and motion. You can easily disable their collisions so they won't interact with the player and won't get him stuck especially if the prop's geometry is complex and not uniform. This is simply done by setting the "collisions" in the props' properties window to "not solid".

Get in the habit of disabling collisions on every small prop that is not critical to gameplay; the best candidates are plants, bushes, flower beds/pots, windows, doors, gutter pipes and every small decoration prop you can think of.

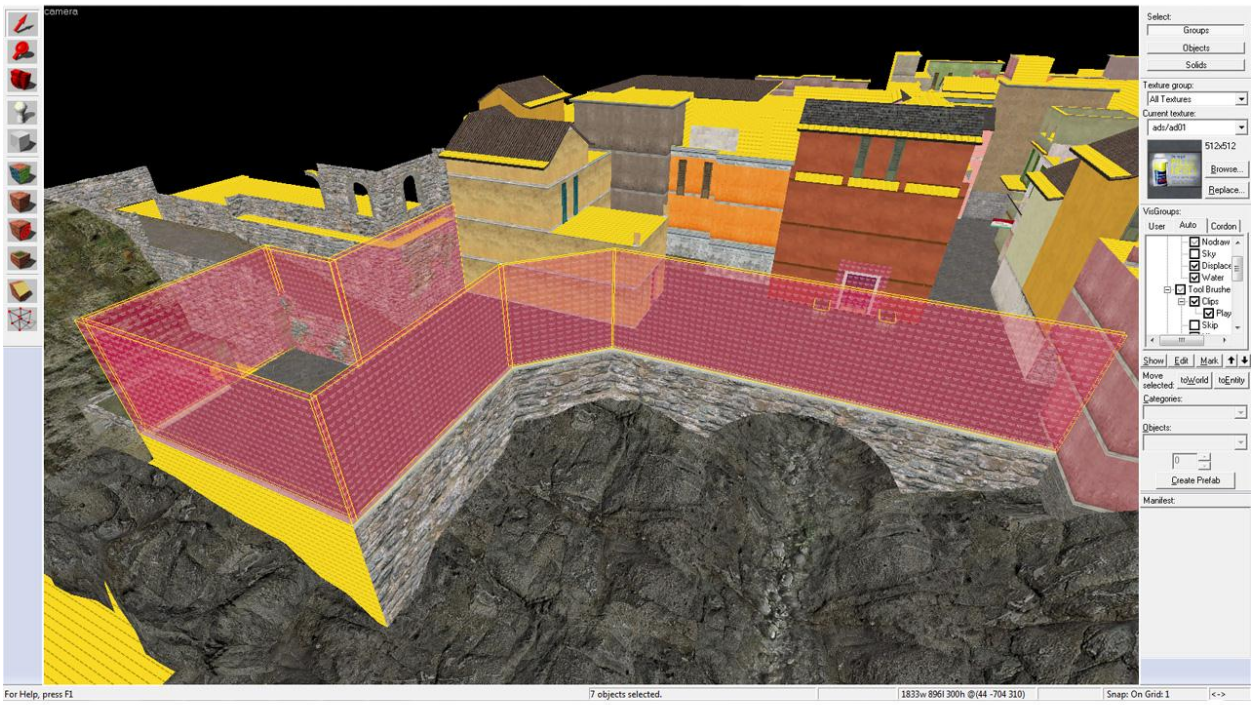
It goes without saying that big props or the ones used to provide cover or strategic gameplay options are not to have their collisions disabled. It would be very silly to have a car or truck as non-solid allowing players to go through them.

Other candidates for disabled collisions are props out of the player's reach. In de_spezia_pro, every prop that is on the balconies, roofs or simply too high for the player to reach, is set to non-solid.

Arches, rails and doors should also be set to non-solid and enveloped with a clip brush allowing for smoother gameplay.

For small prop_physics (bottles, cups, picture frames, small tools and the like), you can set them to be debris, disabling their interaction with the player and acting as if they were non-solid. They will still move when you shoot them or when the bomb explodes but they won't present that bump feeling when you try to interact with them. The debris options can be toggled by checking the "debris" flag in the prop_ physics' properties window, under the flags tab.

Let us see all this in screenshots.



Please note how clip brushes are tracing the contour of CT spawn to prevent players from going outside the playable area. The minimum height here should be 256 units to prevent players from “buddy-jumping” over the clip brush and from using various objects to climb over the clip top. Feel free to go higher until you reach the skybox if you feel that players might take advantage of surrounding geometry to bypass a relatively low clip brush.

As we mentioned before, rails should be set to non-solid and wrapped with a clip brush (labeled 1 in the pic). You can see the clip brush labeled 2 behind it, fully enveloping the door/frame. Notice the angular face just where the number 2 is placed in the pic; this is needed to allow the player to slide over this brush if he is walking close to the door without getting caught on the door frame's pointy edge.

The door is set here to non-solid to prevent the player from getting stuck on its edge (green arrow mark). To counter the effect of the “cavity” created from the door prop being non-solid, I placed a clip brush to be neatly leveled with the door’s surrounding brushes.

Arches also should be set to non-solid. 3 clip brushes will be placed to smooth the player's movement: 1 square in the middle to simulate the lower arch portion and 2 triangular on each side to allow players to slide on the angular face without getting stopped.

As I said in the screenshot of the rails, doors should have a clip brush that is angular on the edges (see arrows locations). This will also allow players to glide smoothly on this surface without interruption.

As we have seen in the door screenshot earlier, all doors that don't open should be clipped with a square brush that fits nicely inside the door's cavity. The doors labeled 1 and 2 are neatly clipped, the corner labeled 3 is also clipped with a triangular brush and the picture frame in between the doors is non-solid; the player will simply glide like a breeze through this corridor without anything tiny on the wall or floor stopping him.

The check clip brushes in-game, enable cheats then type r_drawclipbrushes 1 to highlight all clip brushes in red and all player clips in purple.

This is an in-game shot from bombsite “A” in de_spezia_pro highlighting the location of all clip brushes in the map.

As the title of this paragraph suggests, there is another type of optimization, besides, gameplay to make your level play better and look more professional. Welcome to the final part of this long tutorial: lighting optimization.

What you need to focus on here is lightmaps. A lightmap is a texture generated by vrad during compile, and added to brush faces to simulate lighting. Lightmaps are defined by their scale and the default value is 16; a lower scale (i.e. 4 or 8) will make shadows on this face crispier while higher scales (32 or 64) will render shadows blurry and barely identifiable.

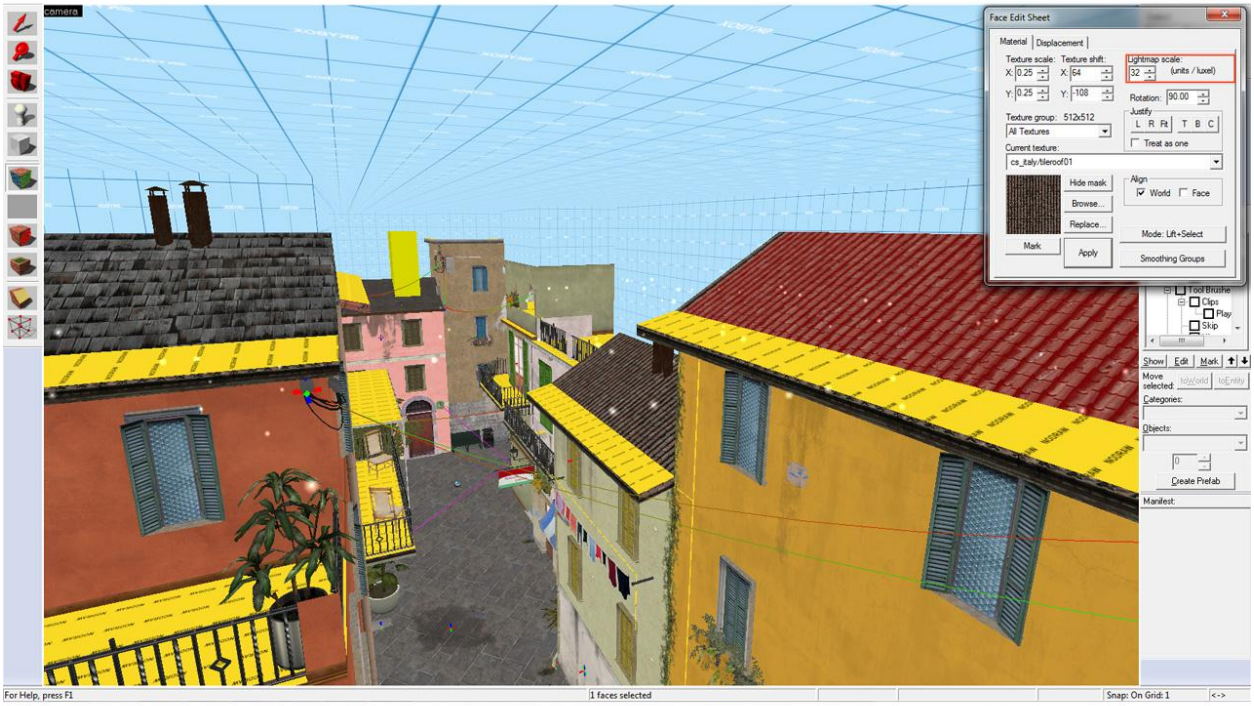
What’s in it for me, I hear you say. Well, let me first show you the pros and cons of each scale level. Low lightmap scale is ideal when used on a relatively small surface, especially indoors when you need to put a certain lighting effect/geometry in perspective. Keep in mind that low scale lightmaps will increase vrad compile time and your bsp size consequently. If you have too many faces with low scale and/or have low scales across large portions of your map, then vrad might even crash due to lack of memory to process all the lightmap textures’ calculations.

And having too many low scale lightmaps might start affecting your in-game performance, because lightmaps are still textures in the end; having too many extra high resolution textures during gameplay might affect the engine processing and rendering of these textures.

Bottom line is to use the low scale lightmap, wisely and scarcely, preferably on small coverage faces and only when needed.

What about high scale lightmaps? The opposite holds true: they will decrease bsp size and vrad compile time, and might shave off some nanoseconds from the engine's processing time that is allotted to them. You still need to be careful not to use them randomly though, as these scales in the wrong place can make your environment look hideous due to the blurry shadows. It is also wise not to have them next to low scale lightmaps as the contrast will be too obvious and will break the realism and immersion in the map.

The prime candidate faces for high lightmaps are roof tops, roof trims and distant parts of the map that are visible to the player but not accessible.



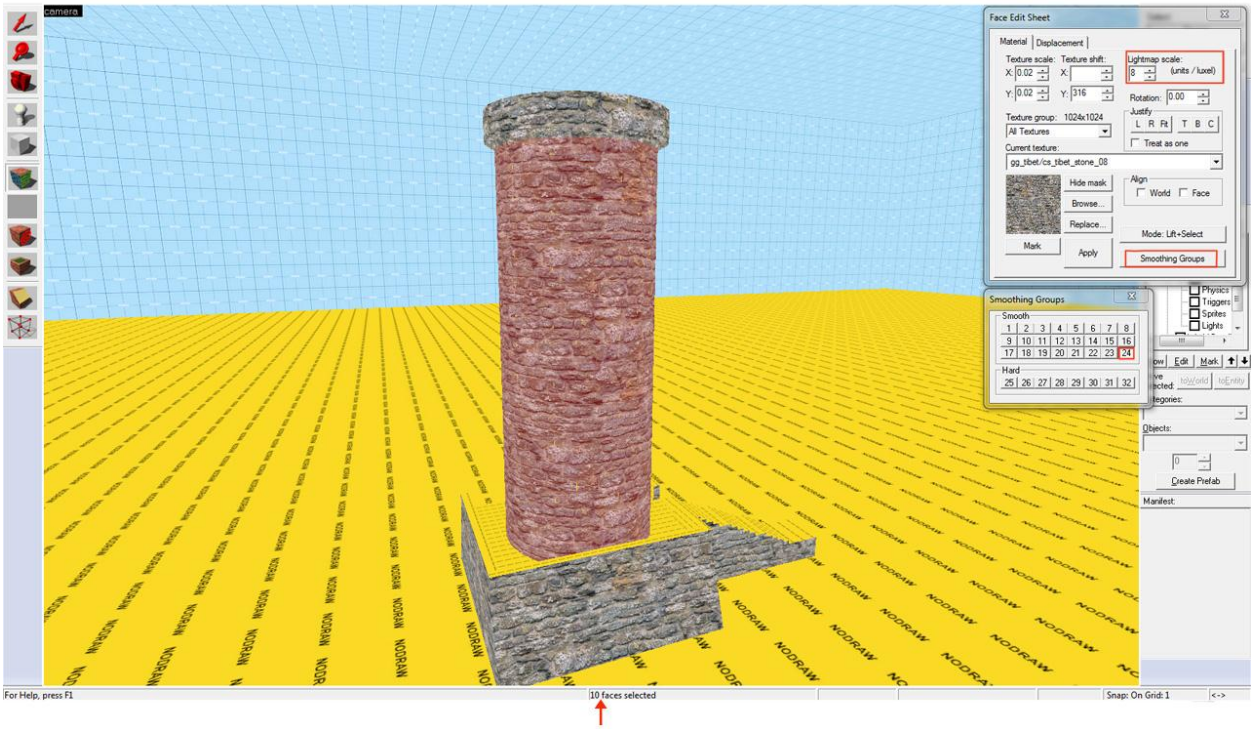
In this screenshot, I have set the scale to 32 for this roof (highlighted red square) through the face edit window (the same window you use to apply textures). You may go up even higher in scale to 64 or even 128 depending on your map and texture. Feel free to experiment with other potential places for high lightmaps; you just need to keep an eye on these faces to make sure no visual glitches are present (you may use mat_fullbright 2 to help you see any lighting anomalies or irregularities on these faces across the map).

The last thing concerning lighting optimization revolves around smoothing groups.

The bsp, hammer and the compile tools are more or less old concepts dating back to the days of Quake and Quake 2 in the mid-nineties. Hammer is still confused with circular and cylindrical shapes when it comes to lighting (and textures alignment, but that's another topic ☺)

In the case of a cylinder, vrad will cast light on the faces that are on the light's side and will ignore the faces that are on the opposite side. The transition between bright and dark is very abrupt and harsh, not to mention unrealistic and ugly. This is where smoothing groups come in handy.

What you are basically doing is instructing vrad to treat the faces of the cylinders (i.e. 12 faces) as one face; this way, light will be cast more homogenously across the faces and the transition of light between each face will be more smooth and diffuse.



This is the tower in the 3D skybox of de_spezia_pro. You just select all the faces of the cylinder (10 faces as denoted in the red arrow location), then click on "smoothing groups" in the face edit window then assign a number for this group in the smoothing groups mini window. You are telling vrad that these 10 faces are group number 24 and they should be treated as one face. If you have other arches/cylinders that need smoothing, then just go through the same steps but at the end, assign a different smoothing group number.

Please note that the lightmap scale that I chose is 8 (upper highlighted rectangle in the pic). This might not be needed and you can sometimes manage the smoothing with the default

scale of 16. However, if you notice that the transition is still buggy and harsh when you test it in-game, then lower the scale of your lightmap (for all 10 faces of the cylinder) to 8.

This is the tower as seen in-game.

The bright side is on the right (facing the light_environment) while the left side is the darker side. You can clearly notice the soft shadows and smooth transition on the cylinder going from right to left. No light glitches or contrast of dark patches near well-lit faces, all thanks to our smoothing groups optimization technique.

IV- In-game Testing and Console Commands

Since you have been patient and read the whole tutorial (didn't you?), I will share with you my secret toolbox of console commands that I personally use to test my maps before release, or to test other people's maps for the purpose of reviewing them and providing feedback.

You already know some of these commands from previous paragraphs while others are new. They shall be grouped here and each one, briefly explained. Don't forget to enable cheats (sv_cheats 1) before using these commands. All these commands can be turned off again by inputting 0 instead of 1 (or vice versa).

- mat_leafvis 1: showcases your visleaves in-game; use it to determine how well your hints are cutting visibility and to check for any visleaves that need further visibility work and

hints placement. Use mat_leafvis 3 to check the PVS where all visleaves that the visleaf you are standing in can see, will be drawn.

- mat_wireframe 1: renders the world in wireframe showing exactly what the engine is rendering in real-time; this is an absolute must to test your areaportals and hints (do not use mat_wireframe 2 or 3)
- r_drawportals 1: highlights all areaportals in the map with a green outline (useful to check their location and to check if you have missed a location without areaportal by mistake).
- r_showenvcubemap 1: turns your hand and weapon model into a bright reflective texture; very useful to test your cubemaps and to see if they are reflecting correctly (wander around the map and the reflections should change based on the nearest cubemap in the area). If the hands turn black (or reflect the skybox only), then you forgot to build the cubemaps or didn't remember to add them to the map.
- r_visocclusion 1: highlights all occluders in the map with a white outline while showing the culled props in green and the visible ones in red.
- r_occlusion 0: turns off occlusion; useful to check the benefits of the occluder versus its cost on engine resources.
- r_lockpvs 1: locks the current PVS; very useful command as it freezes the current PVS you are seeing allowing you to wander around it to see what is being drawn, without updating the PVS as you move. I mainly use it whenever I want to change the location of some hints and areaportals and want to see the benefits (or lack of) from moving them (whether I'm cutting further visibility or making it worse).
- cl_showfps 1: displays your real time fps (frames per second). I don't use it much since I keep net_graph 1 turned on all the time (displays your fps and packets in/out on your lower right corner).
- r_drawstaticprops 0: hides all prop_static in the map
- r_drawfuncdetail 0: hides all func_detail in the map
- r_drawdisp 0: hides all displacements in the map.
- r_drawothermodels 0: hides physics and dynamic props in the map.
- r_drawbrushmodels 0: hides all brush entities in the map. I usually use these last 5 commands together to determine if I have forgotten any details as regular world brush. If you followed this approach in this tutorial, then after hiding all these props, displacements and details, you should be left with only big flat regular world brushes. If you spot any small detail, then chances are that you overlooked it and should go back to Hammer to convert it to func_detail or displacement, or change it to a prop_static.
- r_drawclipbrushes 1: displays all clip brushes in the map; useful to check if you forgot to add clips to certain complex brushes/props in the map.
- r_drawlight 1: highlights the location of all lights in the map; helpful to check if you forgot an area without sufficient lights and to make sure that your lights are distant enough from walls and ceilings to avoid having the bright light patch effect in HDR.

+showbudget: displays your real time frames per second, your map ping in addition to a detailed graph showing the engine cost of each of your level's components (world brushes, props, physics, sounds, vgui, etc.). This can prove very useful if you are experiencing a drop in frame rate at certain locations in the map and you are not sure why it is happening. The highest bar in the graph will let you pinpoint which component is the culprit in sucking engine resources (to turn it off, type –showbudget).

Your optimization effort shouldn't take more than 1-2 days for simple maps and 2 weeks for the more complex ones.

V- Conclusion

As with any systematic approach, reading and understanding the tactic that I showcased in this paper should be your first step. However, to make the most of it, you need perseverance and a lot of practice (test maps are perfect for this end) until this process becomes second nature in you. And trust me; you will get your reward when you see your map having luxurious frame rate and being played worldwide on several public servers.

If you understand this tutorial and follow its methodic progression in your maps' optimization, then you are guaranteed to be amazed by how much your maps would have improved on the optimization front. There are no more lazy excuses for sloppily optimized maps.

If you are an absolute beginner when it comes to level design and mapping, and you are confused by some of the terminology used in this paper, then I strongly advise you to start reading the basics about the various items we discussed in the official Valve developer community wiki (links at the end of this paper).

I also recommend that you read one of my previous papers that was released exactly 1 year ago (funny coincidence) covering a systematic approach to level design (http://source.gamebanana.com/tuts/10747).

I have been in the level design "business" for the past 14 years (since 1998), and as stated in the introduction, optimization was also part of my real life career; these 2 factors were the reason that I wanted to transfer my knowledge in this field to as many aspiring designers as possible in the hope of shedding light on the optimization process that strikes fear in the beginners' hearts and even manages to confuse some veteran mappers.

This tutorial was a big one (and exhausted me in the past 3 months ☺), and I hope that you can use it as a cornerstone to start winning your optimization battles against the Source engine and to further build and improve your optimization skills.

Will2k

March 8, 2013

Contact: **will2k@hot-shot.com**

Website: **http://will8k.tripod.com**

Useful Links

http://developer.valvesoftware.com/wiki/Main_Page Valve Developer Community

Official site of the Source engine wiki and tutorials. Start here

http://www.gamebanana.com/ GameBANANA

Community site with custom content, maps upload and tutorials

http://www.interlopers.net/ Interlopers

Community site with tutorials

My maps that were used for screenshots examples can be found on my website mentioned above.