# Beautiful, Yet Friendly Part 1: Stop Hitting the Bottleneck

by Guillaume Provost

A couple of years ago I was driving home to Quebec when I stopped near the Ontario border to gas up. I got out of my car to stretch and noticed two other travelers engaged in a complicated mish-mash of hand waving and broken English. I approached, thinking I could help the poor fellows by acting as a translator between both parties, when I realized that not only were they both French Canadians but neither of them knew it.

If I found the situation amusing at the time, I've since come to realize that specialized jargons aren't so different from languages as different as French and English. Likc languages, jargon plays a key role in communicating information about a given discipline: but like languages, jargon can also erect artificial barriers between initiates and neophytes.

Game development studios comprise numerous different professions, each with their respective jargons. In the time I've spent bridging the gap between art and code at Pseudo Interactive, it has often been challenging -- but ultimately much more fulfilling -- to try and explain concepts to artists.

As any experienced game artist knows, dealing with a disgruntled graphics programmer mumbling about "vertex diets" isn't the most enthralling part of the day. In our programmer minds, every model should be hand-tuned and lovingly optimized, and every artist should know the intricacies of the hardware and software they are dealing with.

But in the end, a programmer's job isn't to make a good-looking game, it's to empower the artists to make a good- looking game. In this two-part article on graphics performance, I will attempt to explain how to manage performance without losing track of this vision statement. In this month's part, we will take a look at the big picture and explain how you can often solve performance problems without optimizing a single mesh. We will also see when, if, and what to optimize in your scene. In next month's conclusion, we will get down to the details and study how certain modeling practices can improve the performance of your meshes with minimal impact on their visual quality.

## It’s Not Art’s Fault, It’s Level Design’s!

I was happily debugging the other day when an artist dropped by my desk and asked me how many vertices he could pack in a cubic meter without encountering performance problems. Put on the spot, I simply replied, "Ask level design." Although I intended that as a joke, it wasn't exactly bad advice. Programmers can accurately predict how many vertices you can pack into view, but how many "cubic meters" are actually in view at one time is largely determined by level design, not code.

Viewed in this light, level designers operate in a kind of performance macrocosm, and their work provides your best indication of the performance constraints in which you are operating. In fact, on a performance scale, it's a lot easier for things to go wrong in level design than in art. Highly occluded environments that rely on portals and potential visible sets, for example, can suffer greatly from having several portals aligned into view. And instancing a mesh a thousand times in a room isn't likely to be performance friendly no matter how optimized it is.

Balancing level complexity and prop complexity is generally the combined job of the game designers and level designers, but as an artist you'll benefit greatly from knowing in which contexts objects are used. Carefully optimizing meshes can be a tedious, thankless task. Making sure you pay attention to those objects that count most first will get you the most out of the system for the least amount of work.

At Pseudo Interactive, levels are always hashed out first using rudimentary geometry. Once the level has been play-tested and the rough placement of game-logic items has been ironed out, it gets shipped to the art team for beautification. By the time artists start populating a room with art assets, they already know they will have to minimize decorative details if there are, say, 10 highly tesselated bad guys in a room.

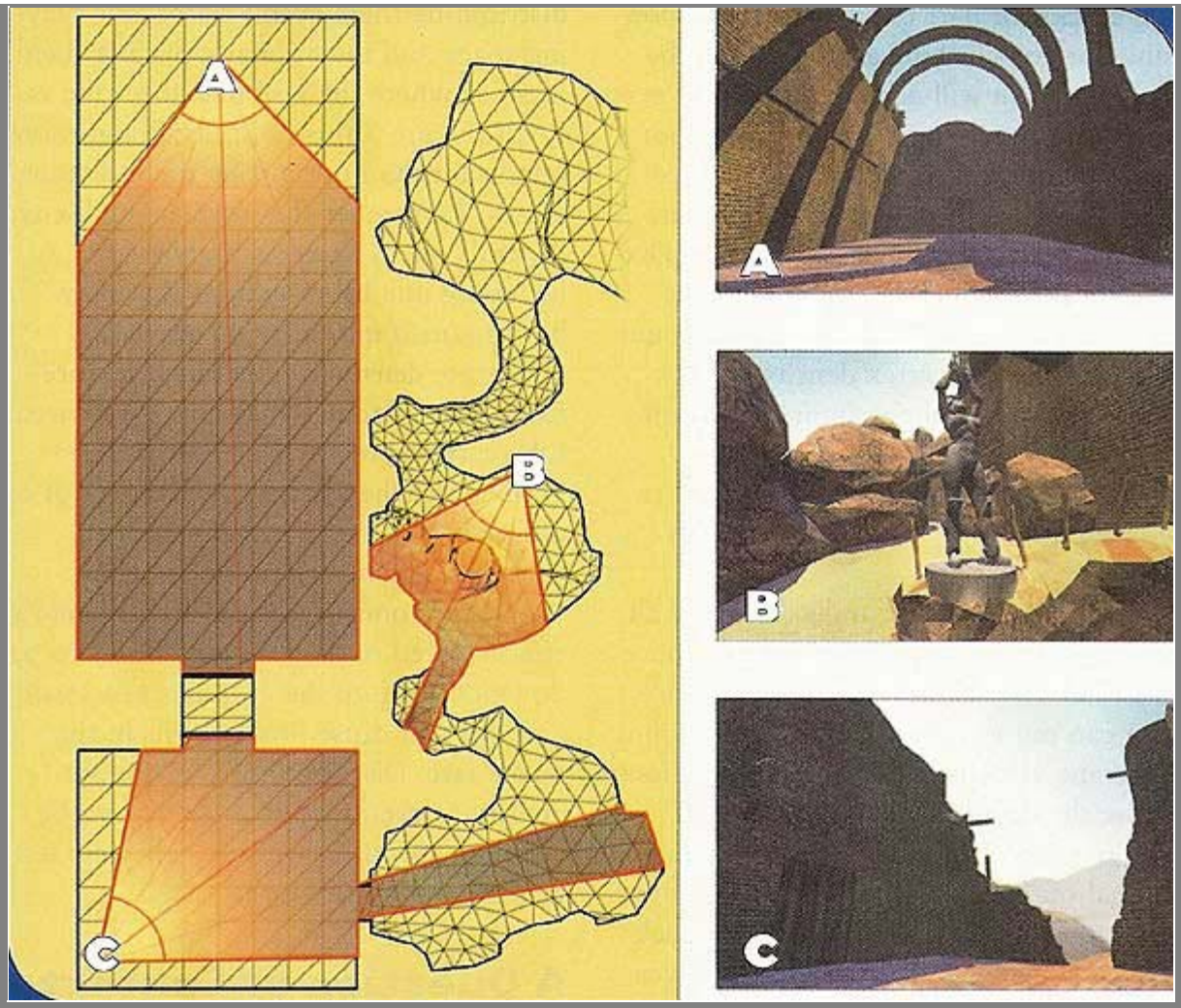



**FIGURE 1.** The visibility spectrum in a portal-enabled visibility engine. In camera A and C, the visibility spectrum is closed off using a door that disables the portal. In camera B, transition zones keep the player from seeing into the next area. The very small visibility spectrum in B lets us place a highly detailed statue that we could not afford in the other areas.

## Murphy's Law

Performance is like laundry: it's a chore, and people only notice it if it's a problem. If your clothes are clean 95 percent of the time, people will only remember -- and judge you by -- those 20 days out of the year where your clothes were not clean.

Frame rate is a direct function of all objects in view at a given time, and as such it is most likely to go down when the visual and environmental stimuli are at their peak. Since, in most games, that also happens to be both when players are most enjoying themselves and when they require the most responsiveness out of the system, performance hits can be significant sources of player frustration.

Provided that you've been somewhat reasonable in building the art assets of a scene, getting good performance out of it is more about avoiding or offsetting the impact of worst-case scenarios than it is about just "making things go faster." No matter what environment you're building art for -- a heavily occluded starship or a snowy medieval landscape -- always assume players will position themselves in the

worst vantage point possible in terms of performance. If there's a single spot in the entire level the player can sit at and bring all its glorious complexity into view ' at one time, you can rest assured that players will strive to get to it.

## Great Wonders Get Separated

There are two increasingly common schools of thought in how the art-development workflow works. In the first, art builds a series of set pieces, and then level design assembles and populates levels using them; this method is not unlike a Lego puzzle. In the second scenario, level design builds basic proxy geometry and populates proxy levels with all logic-related entities, then hands it off to art for embellishment.

But whether you are a level designer constructing a level from existing building blocks, or part of an art team responsible for embellishing a proxy level, the only way you will achieve a constant performance benchmark is to balance your scene complexity correctly. We will see later that there are several factors that may or may not influence scene complexity, but for the purpose of a high-level assessment, you should take into account three things: the vertex density, the texture density, and the visibility spectrum..

***Scene Complexity = Vertex Density * Texture Density * Visibility Spectrum***

The visibility spectrum is the set of all visible space from a given location. Since the total vertex count and texture space you can put into the visibility spectrum is constant, the larger that space is, the less dense the detail you can pack in it. If you're authoring art for a mostly unoccluded outdoor environment, the amount of vertex and texture data you can pack per cubic meter is probably much lower than it could be if you're authoring contents for heavily occluded interior environments (assuming your engine has some form hierarchical visible surface determination system for occluded environments). In fact, the visibility spectrum is the single most important aspect affecting rendering performance.

The smaller you make your visibility spectrum, the more detail you can put in your scenes, and the better your frame rate will be. Typical examples of techniques commonly used to decrease the size of the visibility spectrum include closing off rooms with doors or using transition zones that block off the view for indoor environments (Figures la and Ib), and using fog or depth-of-field effects in external or other typically unoccluded environments.

Vertex density is how tightly packed vertices are in a given volume of space. If small areas of the world contain many highly detailed objects, your performance is likely to drop off when they come into view. If you have a few art assets that are particularly expensive, distributing them evenly across the playing space and intentionally placing them in areas where the visibility spectrum is small (Figure lb) is both faster and more important than optimizing the individual pieces. Remember that scene complexity depends on all assets in a given area, including intelligent entities that may have roamed into it by themselves.

Texture density is how much texture memory you actually use in a given area. Like vertices, textures can be a severe bottleneck when you concentrate a lot of different textures into a constrained location. When such areas come into view unannounced,

all the texture data concentrated in the said area needs to be downloaded to the graphics processor, and this can cause breakdowns in the frame rate. Distributing your texture density as evenly as possible across the visibility spectrum will help alleviate texture-related bottlenecks.

## A Question of Context

So you've examined your level and objects at a high level and fixed what could be fixed, but for gameplay, design, or artistic reasons, you are stuck with certain situations where you need go down to the object level and start making things go faster.

Michael Abrash once said that it's more important to know what to optimize than how to optimize. The rule unequivocally rings true in programming circles, but it applies to graphics-related content just as well as it does to graphics code. If you're running into performance problems, blindly optimizing meshes without any previous evaluation of which ones require it is akin to putting all your clean dishes in with the dirty ones each time you run your dishwasher.

In fact, unless you are not expected to go anywhere near your performance limits -- a common, if dubious, assumption -- there is a subset of your art assets that you should always carefully optimize. If you're working on a third-person game, chances are your main character will get a lot of attention, a lot of detail, and undergo a lot of iterations. Since the character is always in view, its visual quality bar is much higher. But if always being on-screen is a good reason to lovingly fine-tune a mesh, it's also a prime reason to ensure it is performance friendly. Objects that are always or frequently in view are generally the most detailed, most expensive entities in the game, and by the very fact of their presence on- screen, they also constantly eat up processor time. Objects that are instanced a lot may also make their way onto the screen more often than is readily apparent. Always make sure that you carefully balance visual quality and performance when authoring such content.

Since frame rate typically goes down in spurts, common sense dictates that you should be wary of objects that are placed or used in situations that already stress the capacities of the system. Good examples of these include opponents and combat effects. Objects of this nature tend to turn performance problems into performance nightmares, so make sure you identify all stress situations and make their related art assets as performance friendly as possible.

Finally, if your performance problem arises in a specific area, then you will want to hit the most expensive objects in that area first. But how does one properly estimate the cost of an object?

## Transform - Bound, Fill – Bound

The introduction of programmable vertex and pixel pipelines and the new ability of hardware to blend several texture layers in a single pass has made estimating the rendering cost of an object increasingly difficult. I often found myself giving contradictory guidelines to our art team and on several occasions could not formulate clear explanations on what factors came into play and why, until I realized

that, in the end, it came down to a fairly simple set of rules.

The first thing to remember is that graphics processors do not understand or care about objects. They deal with portions of an object on a material-by-material basis. If you want to think in terms of performance, the first thing you should do is to decompose your object into its individual materials, then assess the resulting parts one by one. Programmers call the resulting parts surfaces.

There are two fundamental things that happen when rendering a surface: its vertices get transformed, and its triangles get drawn. At anyone time, both operations happen in parallel, so that whichever of the two is slowest (draw- ing the triangles or transforming vertices) is roughly how fast a mesh can be rendered (Figure 2).

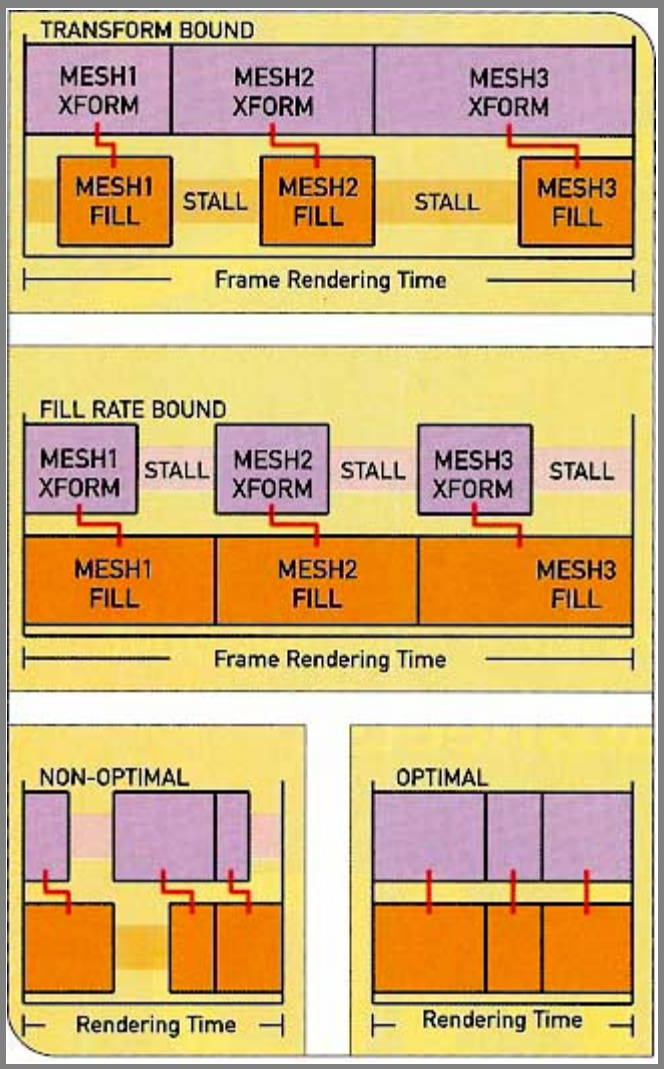



**FIGURE 2.** Transform/fill parallelism.

When assessing a surface's optimization needs, you need to identify both its bottleneck and its cost. The cost will tell you whether you need to optimize it, and the bottleneck will tell you what to optimize. If an object's bottleneck is transform time, then it is said to be "transform-bound," and its cost is the time it takes to transform the vertices attached to it. If an object's bottleneck is fill time, then it is said to be "fill-bound," and its cost is the time it takes to draw the surface on-screen.

Game artists therefore need to develop an intuitive sense of whether a surface is likely to be fill-bound or transform-bound. Fill-bound meshes have different (and

sometimes opposite) optimization rules from transform-bound meshes. So in order to model -- or optimize -- efficiently, one must decide on which set of rules to follow by assessing whether transform or fill is likely to be the bottleneck.

In practice, interactive articulated bodies are generally transform-bound, walls and ceilings tend to be fill-bound, and props lie somewhere in between. But as we'll see, other factors need to be considered when choosing which optimization route to take.

## To Screen Space We Shall Travel

Computational transform cost, the time it takes to transform the vertices, depends on two factors: the transform complexity and the number of vertices to transform. Since programmers like to have nice formulas, I'll attempt to give an abstraction of the trans- form cost here:

***Transform Cost = Vertex Count * Transform Complexity***

The transform complexity is how long it takes to convert individual vertices into screen space. It's highly dependent on what kind of transforms your rendering system supports. In Pseudo Interactive's first title, Cel Damage, we supported (in order of increasing complexity) static objects, binary-weighted bones, seven- point quadratic FFD (free-form deformation) cages, and morph targets.

As a simple example of this, a static wall or ceiling has a very low transform complexity, while a fully articulated zombie with morphing goose bumps has a very high transform complexity. If your renderer can have multiple lights affect an object, then the complexity of the lighting conditions (how many lights, what type of lights) also affects your transform complexity.

The second factor in the equation is vertex count. Surfaces that have a lot of vertices also have a higher total transform cost. If you are authoring content using higher-order surfaces, such as Bezier and B-spline patches, or if you are using hardware-accelerated displacement maps, then your total vertex tally will grow as a function of the tessellation level individual primitives undergo. (All three methods also entail high transform complexity costs.)

Otherwise, surfaces that have very low vertex counts are unlikely to have a high transform cost, unless the transformations they undergo are exceptionally complicated or numerous. Weight-blended morph targets, a very high number of bones, or higher-order FFD cages such as tri-cubic (64 points per cage) and tri-quadratic (27 points per cage) solids would fall into that category. Surfaces that have both high vertex counts and a high transform complexity can easily produce major bottlenecks.

Now, I've talked about transform costs, but not about the likelihood of a surface being transform-bound. A fork lying on a table might contain few vertices, but the cost associated with drawing such a tiny object is so small, that transform might still be the bottleneck. Hence:

***Transform-Bound Likelihood = Transform Cost / Fill Cost***

If the fill cost of a mesh is smaller than the transform cost, an object will be

transform-bound (it will take longer to transform) no matter how small the transform cost is (or no matter how large the fill cost is).

The bottleneck associated with transform-bound surfaces is related to vertices and transformations, so their performance is generally not affected by the nature of their material or the size of their textures (however, on consoles that require vertices to be resent every frame, texture uploads can compete with vertex uploads for bandwidth). Performance can, however, be affected by the number of successive texture stages applied to the surface: for each platform, there is a hard limit on the number of textures it can blend together before it needs to make a second rendering pass. A separate rendering pass entails retransforming all the vertices of the surface, effectively doubling the transform cost associated with the mesh.

Mainstream consoles and PC video cards support anywhere between one and eight textures per pass. Since the number of texture passes a material requires is highly specific, you should consult your programmer to find out when you are causing extra passes to occur.

## Fat Triangles Make for Fat Fill Times

So what about your average ceiling or wall? Fill time (or draw time) depends largely on the size of the surface on-screen, the number and size of textures involved, and the draw complexity:

***Fill Cost = Pixel Coverage * Draw Complexity * Texture Density***

The draw complexity is the complexity of the operations that occur every time a pixel gets drawn. It is typically a function of how many texture passes are involved and what kind of mathematical calculations occur with respect to those passes. In general, sophisticated per-pixel lighting effects, such as bump and normal maps or spherical harmonics, tend to have high draw complexities that grow increasingly complex with the number of lights by which they are affected. Materials that distort the view, such as refractive glass, or materials that cast volumetric shadows, also have very high fill costs. Since you can often combine several such properties on a single surface, the fill cost of multi-pass surfaces can go through the roof.

Even if your draw complexity is pretty tame, it's easy to forget the potential bottleneck caused by texture size and density. A wall filled with a giant mural, a high-resolution light map, and a detail map to dirty it up might screw with your texture cache because of the high volume of texture memory to which it refers.

No matter what, your surfaces are generally unlikely to have high fill costs if they are small on-screen. But since triangle sizes are pixel-based, higher resolutions or FSAA (full-screen anti-aliasing) modes will directly affect your fill rate by enlarging every triangle's pixel area. The higher the resolution, the more important fill rate becomes. In fact, enabling FSAA is a very common technique to estimate roughly how much of a scene is fill-bound.

Surfaces taking up a lot of screen space with either high draw complexities or a lot of texture data can be major bottlenecks. Although you can't exactly make your walls and ceilings any smaller than they are, you should be wary of the material properties

you set on those surfaces.

The fill-bound likelihood is the inverse of the transform-bound likelihood:

***Fill-Bound Likelihood = Fill Cost / Transform Cost***

If the transform cost of a mesh is smaller than the fill cost, an object will be fill-bound (it will take longer to draw) no matter how small the fill cost is (or no matter how large the transform cost is). In any case, if both transform cost and fill cost are low, you should skip the object and concentrate on something more problematic.

You may have noticed that the expansion of the fill-bound and transform-bound likelihood equations promptly throws us into programming geek-world. We can abstract it further by extricating the two most significant components out of the equation: vertex counts and pixel coverage. This gives us vertex density, which is really just the vertex count divided by the screen size of an object.

***Transform-Bound Likelihood = Vertex Density***

Although this is an extremely simplified abstraction of the fill-bound, transform-bound question, it makes approximating the answer somewhat more manageable: screen vertex density is your best -- if not sole -- indicator in distinguishing transform-bound surfaces from fill-bound surfaces.

## From Fat to Flat and Big to Small

As an artist, it can be difficult to predict the on-screen vertex density of an object during play: as objects get farther into the distance, the on-screen vertex density rises. This can cause large variations in vertex and texture densities to occur when objects are viewed at different distances from the camera (Figure 3).

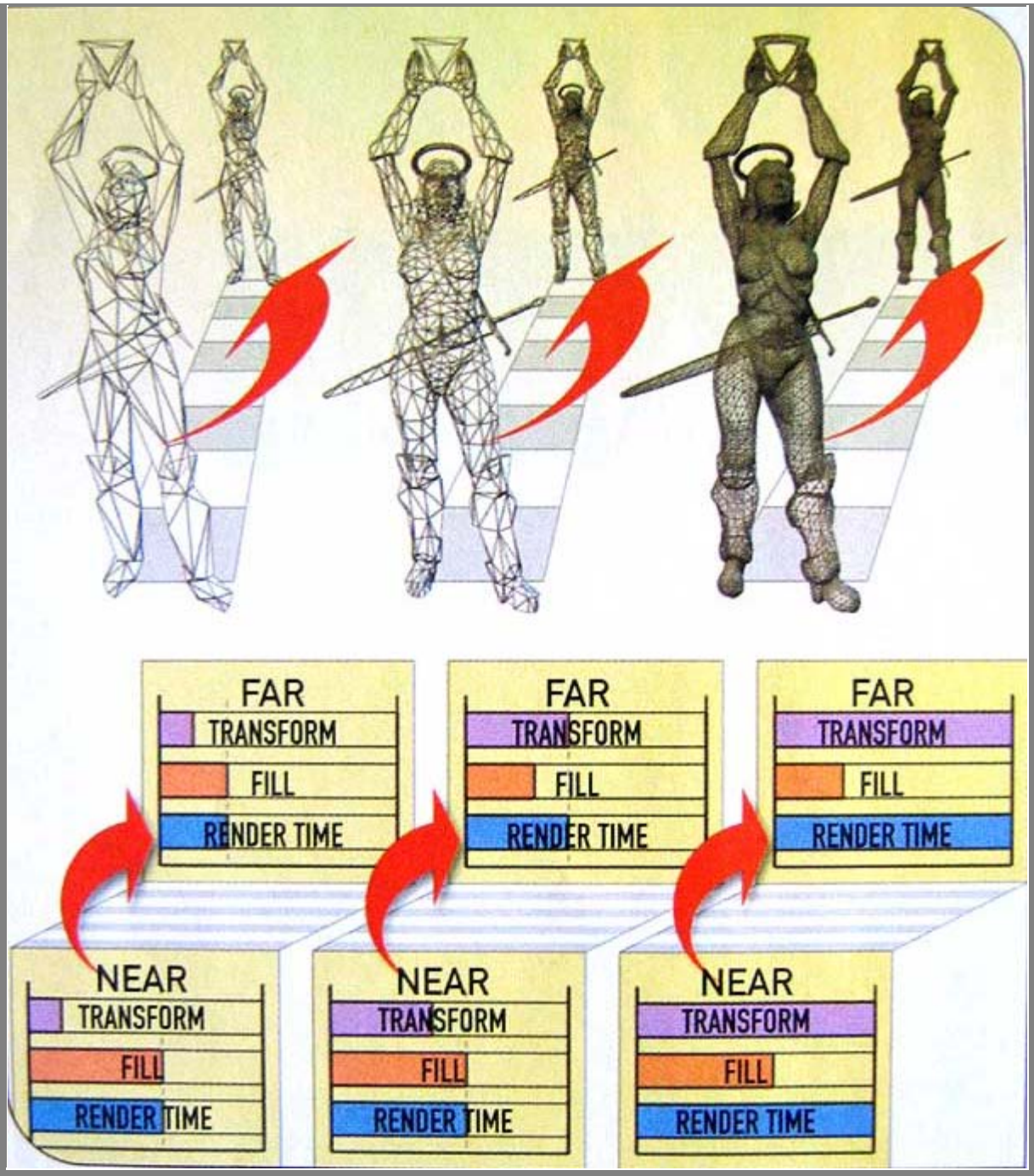



**FIGURE 3.** The fill cost of a mesh decreases with distance, while the transform cost does not. This can cause meshes to shift from being fill-bound to being transform-bound as they recede into the distance (center). If a mesh has a very high vertex density, then its cost may stay the same regardless of distance (right). To reduce the associated transform cost of a mesh in conjunction with its diminishing fill costs, lower-level-of-detail meshes are used (left).

There are two very important techniques that exist to equilibrate an object's vertex and texture density through distance metrics: discrete levels of detail, and texture mip-maps. Discrete levels of detail (or LOD meshes) replace the full-detail versions of a mesh when it is sufficiently far away from the viewer that the switch produces negligible differences in the rendered image quality. Similarly, mip-maps are lower-detail versions of a texture that can be used when the texel-to-pixel density is sufficiently high. Well-constructed mip-maps will actually enhance your image quality and are an absolute prerequisite to making fill-bound surfaces performance friendly.

Although both mip-maps and LOD meshes tend to have a greater impact in outdoor environments, where the very large visibility spectrums call for techniques to minimize the impact of high-detail objects, they both are important to learn and use in all situations.

Skimming vertices and textures out of your mesh will never hurt. But, as we'll see next month, optimizing a mesh for best performance is not a piece of cake. So if you're going to go to lengths to make certain objects truly performance friendly -- and in certain cases, you should -- then you should build at least one proper level of detail for them first.

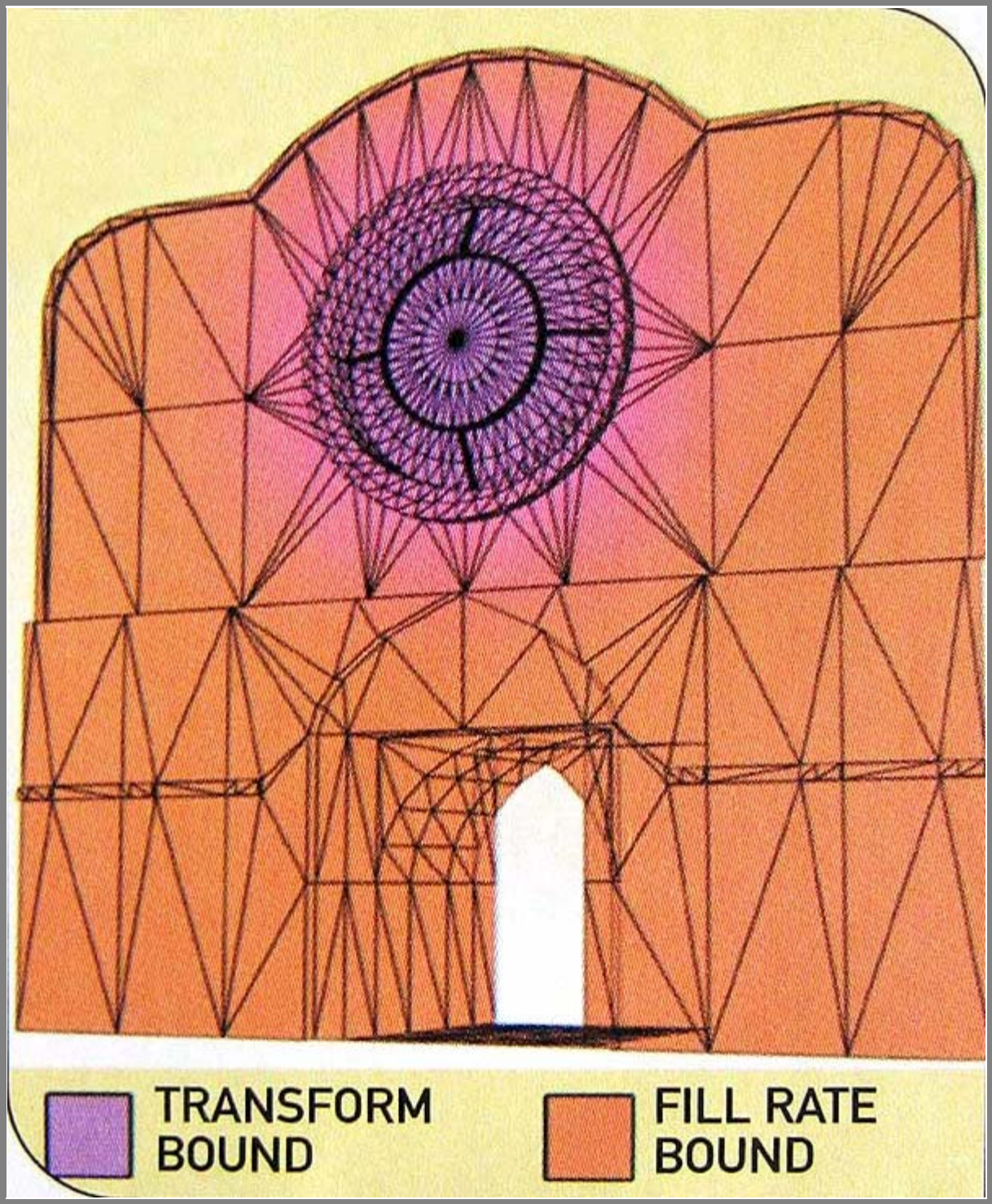



**FIGURE 4.** Non-uniform density mesh.

Finally a surface can also cycle between being transform-bound and fill-bound if its vertex density is very non-uniform (Figure 4). Surfaces with both very high and very low curvature areas or surfaces that are adaptively tessellated for lighting conditions typically suffer from this problem. When dealing with such surfaces, remember that it's important to save vertices in the high-density areas, not in the low-density areas where the renderer is likely to be fill-bound. Better yet, try to distribute your vertex density as equitably as possible across its surface area, balancing the load between transform and fill, and maximizing your use of both processing pipelines.

## When Big Is Too Big

There's an exception to this whole system: objects that are always only partly visible.

If you were to merge all rooms and corridors of a level into a single object, its resulting vertex density might be low, but you would only ever draw a very small portion of it at every frame. When a surface comes into view, only the visible portion needs to be drawn, but all its vertices need to be transformed and all its textures need to be sent to the card.

If -- and only if -- you have objects that have either a significant amount of vertices or large amounts of texture data, and if those objects are too big ever to be entirely in view at once, then you should break them up into smaller pieces. If you do break an object up because of its size, try splitting it in roughly equally distributed volumes to maximize culling efficiency. If your object has a great deal of different materials, try to break up the object in a way that balances the texture load across chunks.

Last week, our art staff built a sky box. The texture they applied on it was -- quite understandably -- very detailed. Since its refined color gradations did not palletize very well, we were suddenly stuck with a 512K texture gobbling up almost all of our available texture memory every frame. We correspondingly split the sky box into four sections, so that at any one time, only half of the original texture was required on-screen, unless you looked straight up. Since there isn't all that much to render when you look straight up, splitting in this case was the right call.

## Microscopes and Binoculars

You'll avoid many performance headaches by paying attention to context and scale. If certain props are consistently located far away from the in-game camera, then you should naturally give them less detail.

Similarly, chances are that small objects will be small on-screen too. If you model objects without a scale reference, you are more likely to spend your vertex budget on a scale where the detail will be lost on the player.

## Next Month

You've analyzed your scene, made reasonable distribution decisions, and identified certain objects as needing an optimization pass. You found what bottlenecks and costs were likely to be -- but then what? And how does this all affect day-to-day

work? In next month's conclusion, we will explore the hands-on modeling and texturing techniques that can be used to reduce cost for both transform-bound and fill-bound surfaces.

***Acknowledgements***

*Thanks to Danny Oros for providing both the illustrations and several rounds of feedback. Thanks also to Benoit Miller from Matrox, Sebastien Domine and Sim Dietrich from Nvidia, Guennadi Riguer from ATI, and Ryan McLean from MagiTech, for kindly answering questions and reviewing the article. And last but not least, thanks to the rest of the Pseudo team, for playing the guinea pigs on this article.*

***Guillaume Provost***

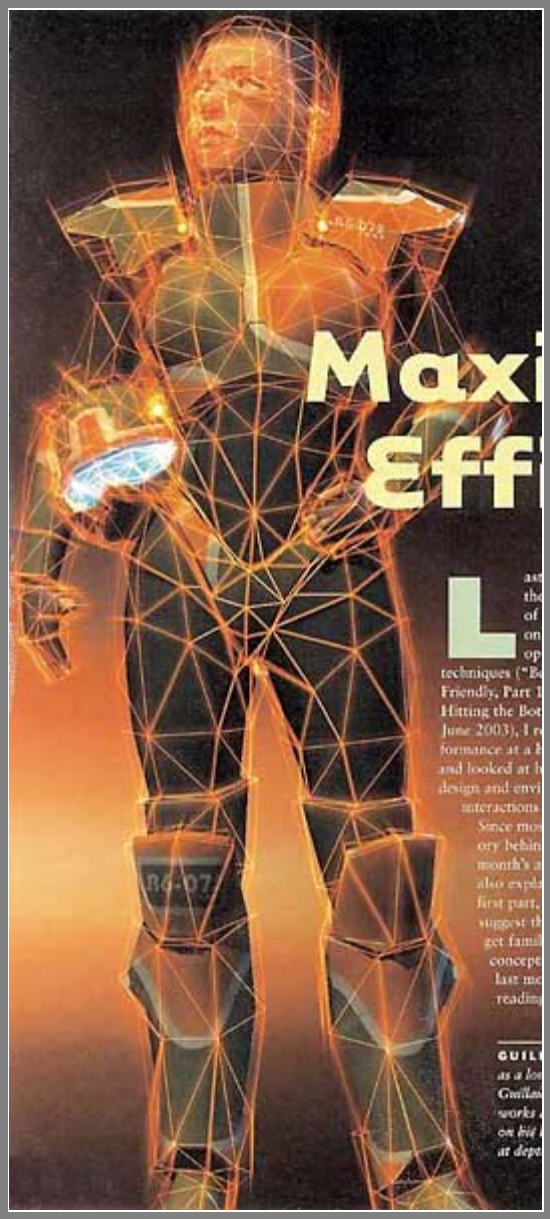
*Originally hired at the age of 17 as a lowly systems programmer writing BIOS kernels for banks, Guillaume has been trying to redeem himself ever since. He now works as a 3D graphics programmer at Pseudo Interactive and sits on his local Toronto IGDA advisory board. You can contact him at depth (at) keops (dot) com.*

**Beautiful, Yet Friendly Part 1: Stop Hitting the Bottleneck**
**Beautiful, Yet Friendly Part 2: Maximizing Efficiency**
*Articles originally published in Game Developer magazine, June and July 2003*

Beautiful, Yet Friendly Part 1: Stop Hitting the Bottleneck
Beautiful, Yet Friendly Part 2: Maximizing Efficiency

# Beautiful, Yet Friendly Part 2: Maximizing Efficiency

**by Guillaume Provost**



Last month, in the first part of this series on content optimization techniques, I reviewed performance at a high level and looked at how level design and environmental interactions affect it.

Since most of the theory behind this month's article was also explained in the first part, I strongly suggest that readers get familiar with the concepts introduced last month before reading this article.

You'll need to know when and what to optimize before you can make any use of knowing how to optimize.

Last month, we saw that meshes could be transform-bound or fill-bound. I've given a more complete picture of the possibility space here through the generic hardware pipe shown in Figure 1.

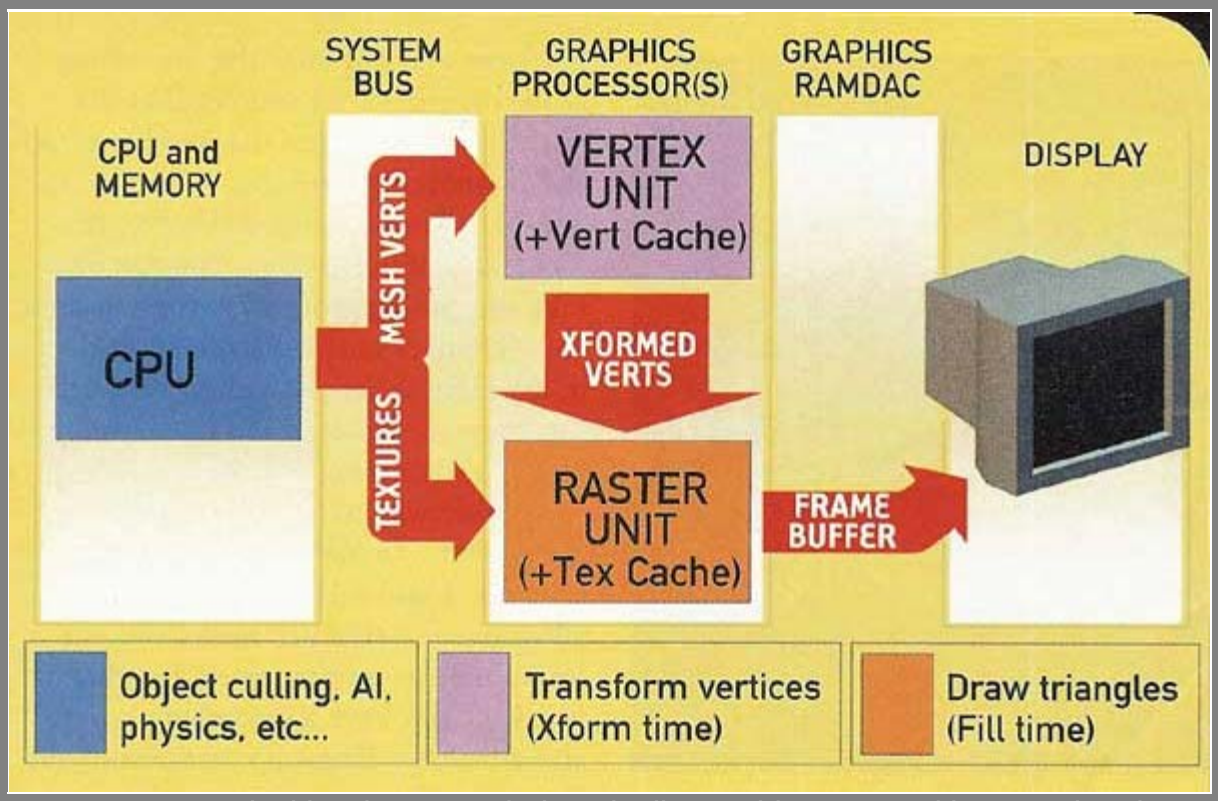



***FIGURE 1.*** *A typical hardware rendering pipeline architecture and its associated bottlenecks. Typical bottleneck scenarios:*
*1. Transform bound. The vertex unit can't transform fast enough.*
*2. Fill bound. The raster unit can't draw the polygons fast enough.*
*3. Data bound. The bus can't ferry all the data fast enough.*
*4. CPU bound. The CPU has to cull too many objects, and/or is clogged by other game-logic-related tasks.*

If you are data-bound, then the amount of data transferred might also be causing transform problems (too many vertices) and/or fill problems (too much texture data). Data-related problems generally arise through a collection of objects, not by single objects in isolation. If you find that you're clogging the bus -- generally when there's too much texture data -- then you should redistribute your texture and vertex densities across your scene (last month's article described how to do this). If you are CPU-bound, then it's out of your hands; the programming team will need to take a hard look at their code.

## Optimizing Transform-Bound Meshes

If design wants marching armies of zombies attacking the player, you'll need to make sure they don't put the renderer (and artist) on death row by minimizing their transform cost.

We saw last month that the cost of a transform-bound mesh is:

*Transform Cost = Vertex Count * Transform Complexity*

Hence, we need to reduce the transform complexity or the number of vertices. You can somewhat reduce the transform complexity by plucking out bones you don't really need, but you should consider using a less expensive type of transform first. If you can approximate a morph target accurately enough with a few bones, you'll save on transform complexity. If your engine is optimized for nonweighted vertex blending (where vertices can be affected by only one bone), see if you can substitute your vertex-weighted mesh with a clever distribution of bones that take no vertex weights. In any case, take the time to consult with the programmers, as they may have insights on better transform techniques you can use to lower your transform complexity.

## Welcome to Splitsville

Before you go plucking vertices out of your mesh, I'll let you in on a secret: the vertex counts in your typical modeling package don't reflect reality. As they travel down the pipeline, vertices get split and resplit ad nauseam. Vertex splits adversely affect transform-bound meshes by adding spatially redundant vertices to transform. In theory, vertices can get split as many times as they touch triangles, but in practice, total vertex counts generally double or triple. Keeping this in mind, you can lower this split ratio dramatically and make your mesh a whole lot more performance-friendly without removing a single vertex.

Let's first examine the nature of the splits. As I mentioned last month, graphics hardware thinks in terms of surfaces, not objects (that is, the set of all faces in an object that share the same material properties). So the first vertices that get split are those lying on the boundaries of two different surfaces. Think of it in your head as: A vertex cannot be shared across multiple materials (Figure 2b).

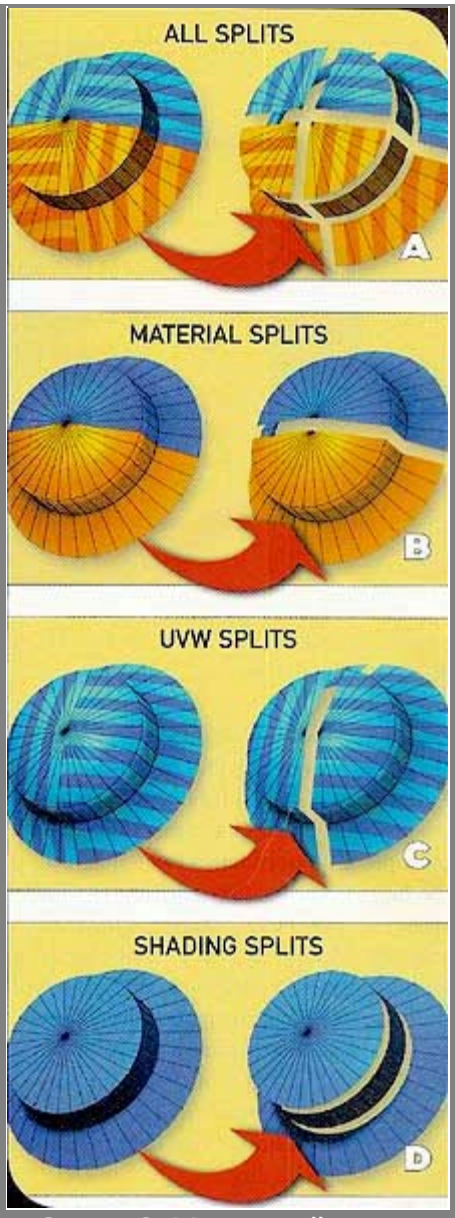



***FIGURE 2.*** *Vertex splits accumulate over UV discontinuities, smoothing group boundaries and material boundaries.*

Similarly, renderers typically do not allow vertices to share polygons with different smoothing groups, or vertices that have different UV coordinates for different triangles. So vertices that lie on the boundaries of two different smoothing groups are split, and vertices that have multiple UV coordinates (which lie on the boundaries of discontinuities in UV space) will also cause splits (Figures 2c and 2d). Moreover, if you have objects with multiple UV channels, the splits will occur successively through every channel.

There are several simple ways to minimize individual types of splits. Intelligently combining and stitching textures together, for example, can help minimize material-based splits.

UV space discontinuities tend to be a bit trickier. Mapping an element without any UV break means that you'll have to find either an axis of symmetry or at the very least a "wrapping point" on your mesh.

If you can get away with using mapping generators, such as planar, cylindrical, or

cubic mappings, you can minimize or altogether eliminate UV space discontinuities. Ball-jointed hips and shoulders, for example, can make the resulting arm and leg elements ideal candidates for such techniques.

If you need to split the mesh in UV space, both 3ds Max 5 and Maya have elaborate UV-mapping tools that permit you to stitch UV seams in order to minimize the damage (Maya even has a UV- space vertex counter, which should reflect the number of vertices in your mesh after UV splits.). It's generally well worth spending the time to optimize your mapping in UV space, since it will also both simplify your texturing pass and minimize the texture space you will actually need for the object. When no axis of symmetry existed, we found that treating the texture as pieces of cloth that you "sew" up worked well to minimize UV splits when texturing humanoids (Figure 3).

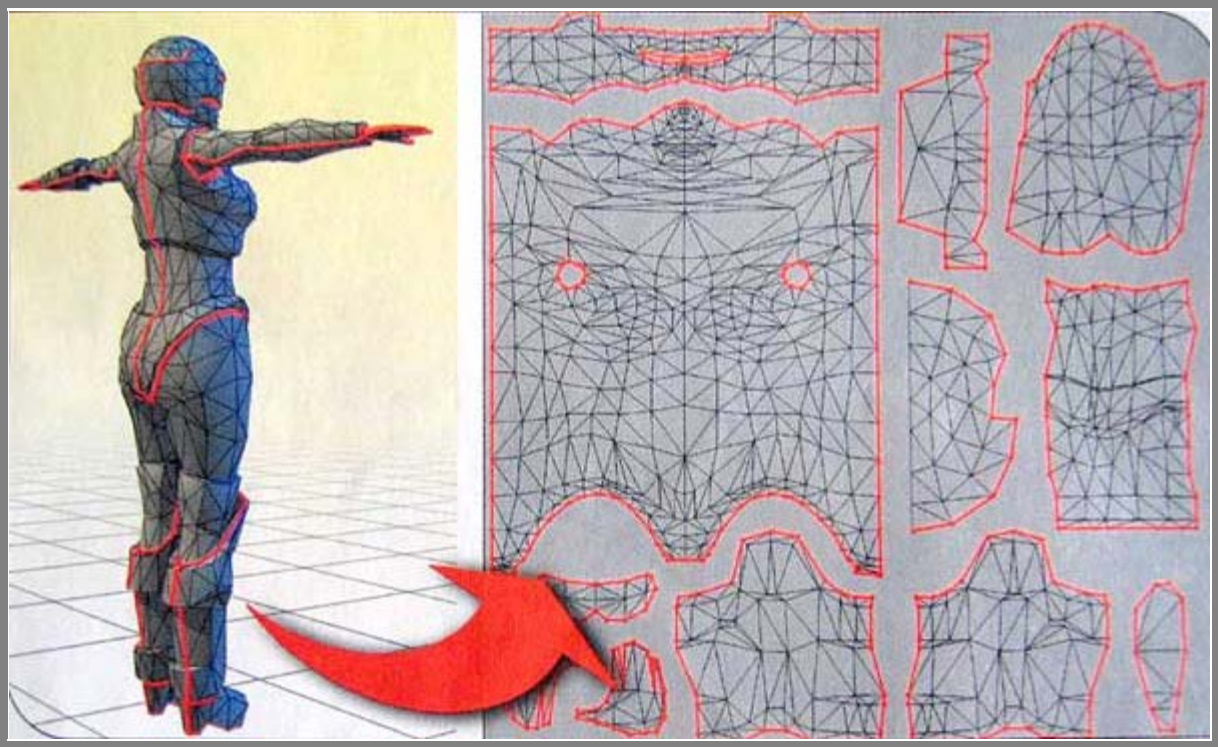


***FIGURE 3.*** *The texture is unwrapped on the mesh along a single "sewing" line that wraps the texture like a piece of cloth. This minimizes UV discontinuities (shown here in red) without introducing constraints in the visual look of the mesh.*

If you are building a performance-practical mesh, it's probably best that you fine-tune and optimize the smoothing groups by hand. Remember that the goal isn't to minimize the number of different smoothing groups, but rather the number of boundaries that separate those smoothing groups. You can also fake smoothing groups by using discrete color changes in the texture applied to it, avoiding splits altogether, although this may not result in the visual quality you are attempting to achieve.

Another way to look at it in the big picture is to "reuse" vertex splits. For example, I said earlier that renderers allow one material per vertex and one smoothing group per vertex. In other words, if you have a smoothing group and a material ID group that occupy the same set of faces, they'll get split only once. The same goes for UV discontinuities: if they occur at smoothing group boundaries, then they won't cause an extra split to occur.

For the record, if your mesh is definitely transform-bound, then it is generally more important for you to save on vertex splits than to save on texture memory. If that means authoring an extra texture for the mesh in order to get rid of individual diffuse color-based materials or UV breaks, then it's a fair trade-off.

This brings us to normal maps and the general (and increasingly popular) concept of using high-detail meshes to render out game content. Normal maps are textures for which every texel represents a normal instead of a color. Since they give extremely fine control over the shading of a mesh, you can replicate smoothing groups and add a whole lot of extra shading detail by using them. Since normal maps are generally mapped using the same UV coordinate set as the existing diffuse texture, they do not cause extra vertex splits to occur, and are in effect cheaper for transform-bound meshes -- and much better looking -- than smoothing groups.

Unfortunately, normal maps cannot really be drawn by hand; they require specialized tools to generate them, and also require higher-resolution detail meshes if you want to take full advantage of their potential. Because of the pixel operations involved that are required to support them, they are also not supported on all hardware platforms.

Overall, absolutely try to avoid checkerboard-like material switches, where you consistently cycle between materials. Unless your programmers specifically support it, also avoid setting whole objects as flat-shaded by having individual faces each be a different smoothing group (Figure 4).

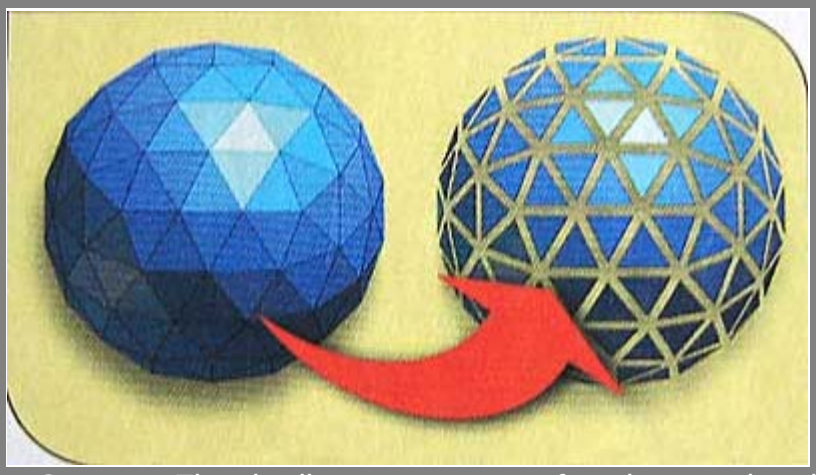


***FIGURE 4.*** *Flat shading causes every face in a mesh to belong to a separate smoothing group, causing a worst-case split scenario to occur. Avoid at all costs unless specifically supported.*

## Helping the Stripping Process

When I originally set out writing this article, I naively thought I could safely cover solid guidelines that covered all mainstream console systems and all recent PC-based graphics cards without encountering critical system-specific guidelines. I was overly optimistic.

Some systems don't support indexed primitives, and some don't have a T&L transform cache. In either case, your surfaces' transform cost will be significantly affected by their "strip-friendliness." If your hardware does support both, then strip-friendliness is less of a performance issue.

A triangle strip is a triangular representation some systems use in order to avoid transforming a vertex multiple times if it's shared among one or more triangles. In a triangle strip, the first three vertices form a triangle, but every successive vertex also forms a triangle with its two predecessors. When graphics processors draw these strips, they only need to transform an additional vertex per triangle, effectively sharing the transform cost of the vertices with the last (and next) triangle.

Stripping algorithms close a strip (effectively increasing transform time) when there are no vertices they can choose in order to form a new triangle. This typically happens at tension points (Figure 5), where a single vertex is shared amongst a very high number (eight or more) of triangles. (Certain renderers support what are called triangle fans. Fans make tension points very efficient, but given that current hardware only supports one type of primitive per surface, they tend to rarely be supported in practice.)

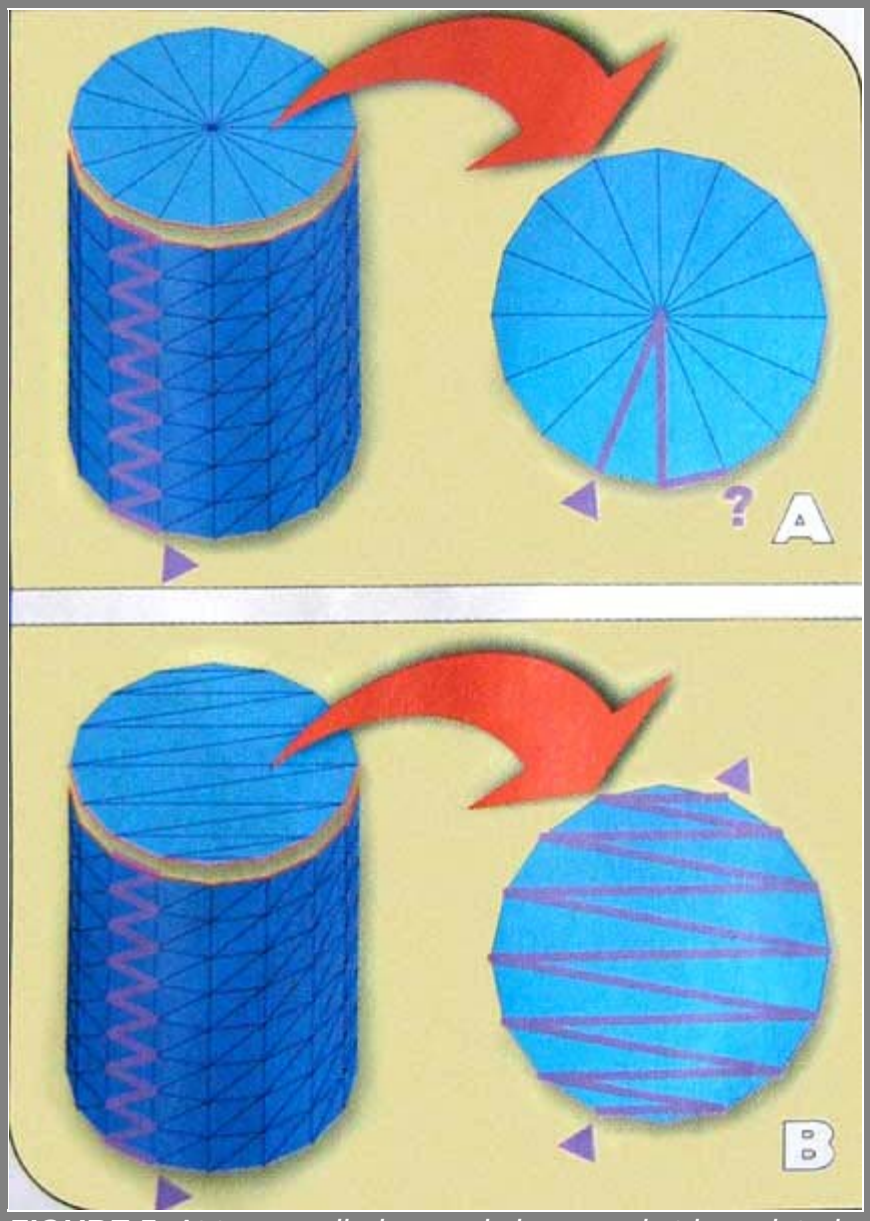



***FIGURE 5.*** *At top, a cylinder cap is improperly triangulated, causing the strip to "break" very early. The strip cannot cross to the main body of the cylinder because of smoothing group splits. At bottom, the cap is retriangulated properly, and fits completely in a single strip.*

Since tension points are always connected to a series of very thin triangles, avoiding sliver triangles and distributing your vertex density as equally as possible on the surface of your mesh will generally help the stripping process.

Most good triangle-stripping algorithms will automatically retriangulate triangles lying on the same plane, but they cannot reorient edges binding faces on different planes. You should verify these details with the programmers.

## Transform-Bound Meshes Conquered

Knowing about all these technical details can make a transform-bound mesh up to three times more efficient if you're smart about what you're doing, but it's still a lot of work. Always ask yourself whether you need to optimize a mesh before you dive into the hard work. Otherwise, use these techniques opportunistically. In the end, having a tool that helps visualize where vertex splits occur is tantamount to building truly optimized meshes. As a summary of things to look out for, here's an optimization checklist for transform-bound meshes:

- Build one or more LOD (level of detail) meshes for the object.
- Use as few bones and vertices as you can, and try to decrease the transform complexity.
- Use as few material surfaces as you can get away with; consider texturing your mesh instead of using several different diffuse colors.
- Use UV generators to minimize UV discontinuities.
- Get rid of smoothing group breaks you don't really need, or use discrete color changes to fake them, or use a normal map.
- Match the remaining material boundaries, UV-space boundaries, and smoothing group boundaries.
- Validate your invisible edges and look out for tension points.
- Avoid sliver triangles and try to make the vertex density as uniform as possible across the surface of the mesh.

If you think that your mesh is fill-bound instead of transform-bound, then do not do any of the above. Combining materials into a single texture applied to a fill-bound mesh, for example, might actually hurt your performance by causing cache misses to occur more frequently, so fill-bound meshes warrant separate optimization considerations.

## Optimizing Fill-Bound Meshes

We saw earlier that the cost associated with drawing fill-bound meshes was a function of three things: *Fill Cost = Pixel Coverage * Draw Complexity * Texel Density*

You can't make your walls any smaller than they are, but you should avoid overlaying several large surfaces within the same visibility space. A typical example of this would be to have an entire room's wall covered with an aquarium (the back wall and the glass window create two layers), or successive sky-wide layers of geometry to simulate a cloudy day. Transparent and additive geometry tend to accumulate on-screen, potentially creating several large layers of geometry the

renderer needs to draw, thereby creating a fill-related bottleneck.

If your export pipeline supports double-sided materials, be wary of using them arbitrarily on large surfaces; you can easily double your fill-rendering costs if you are forcing the render to draw wall segments that should be culled. On some platforms, back-face culling is not an integral part of the drawing process, and culling individual polygons becomes a very expensive task; if you are authoring content for such platforms, you should ensure that walls that don't need back faces don't have them.

The bigger the triangles, the less texture space you want to address. Unfortunately, in practice, meshes that take up the largest portion of screen space also tend to also gobble up the most texture space, and so they are prime targets for being fill-related bottlenecks. There are two things you should do to minimize your texture space: make sure you are using and generating mip-maps, and choose your texture formats and size intelligently.

Table 1 illustrates savings you can achieve by making smart choices about your texture formats. Note that if your textures are smaller than 32x32 texels, it's probably not a good idea to palletize them, since the cost associated with uploading and setting up the palette is larger than just using the unpalletized version. If your hardware supports native compression formats, such as DXT (DirectX Texture Compression), it's a good idea to use them over palettes.

| Size/Format | 16-color PAL | 256-color PAL | 16-bit RGB | 32-color ARGB | DXT1 RGB |
|---|---|---|---|---|---|
| 32 X 32 | Do not use | Do not use | 2K | 4K | 1K |
| 64 X 64 | 2K | 4K | 8K | 16K | 4K |
| 128 X 128 | 8K | 16K | 32K | 64K | 16K |
| 256 X 256 | 32K | 64K | 128K | 256K | 64K |

***TABLE 1.*** *This table illustrates simple savings you can do by making smart choices about your texture formats.*

If you can get away with using diffuse colors only on a fill-bound surface, so much the better. On several platforms, drawing untextured surfaces is faster then drawing textured ones.

I mentioned earlier that it was generally a fair trade-off to sacrifice texture space in order to prevent UV splits in transform-bound meshes. When your mesh is fill-bound, however, the contrary rule applies: if splitting the vertices in UV space will help you save texture space, it's also a fair trade-off.

Finally, conservative decisions on the nature of the materials you apply to fill-bound meshes payoff in performance. The number of texture passes and the complexity of their material properties is always the biggest factor at play when dealing with fill-bound surfaces.

## Texels Miss the Boat

Some of us deal with the creme de la creme when it comes to hardware, but the vast majority of us need to contend with market realities. In the console market, teams

get to push a system to its limits, but they are also stuck with those limits for a long time.

If you count yourself in that situation, then chances are you need to take something called texel cache coherency into account. Here's how it works.

Graphics processors typically draw triangles by filling the linear, horizontal pixel strips that shape them up in screen- space. Almost all current hardware can do this by "stamping" several pixels at a time, greatly decreasing the time it takes to fill the triangle.

For every textured pixel the card draws, it needs to retrieve a certain amount of texels from its associated texture (since the pixels are unlikely to fall directly on a texel, renderers typically set up video hardware for bilinear filtering, which fetches and blends four texels for each texture involved). It does this through a texel cache, which is basically a scratchpad on which the card can paste texture blocks. Every time the card draws a new set of pixels it looks into its cache. If the texels it needs are already present in the scratchpad, then everything proceeds without a hitch. If some texels it needs are not in the cache, then the card needs to read in new texture chunks and place them in the cache before it can proceed with drawing. This is called a texture cache miss.

A good texel cache coherency means few texture cache misses occur when drawing a surface. A bad texel cache coherency will significantly increase the time it takes to draw a surface. Most PC-based systems and a few of the current high-end consoles will automatically ensure a good texel cache coherency by choosing the proper mip level at every pixel they draw. But other systems rely on the fact that the texel density across the surface area of a mesh in geometric space is constant for their mip level choice to be correct.

On such systems, non-uniform texel densities will cause the card to "jump" in texture space from pixel to pixel. This can cause severe texture aliasing problems and will also consistently cause texture cache misses to occur as the card tries to fetch texels that are not in its scratchpad.

As an artist, you can solve both those visual artifacts and performance problems by ensuring you uniformly distribute texel density across your mesh (Figure 6). You can do this by ensuring that the size and shape of your faces in UV space is roughly proportional to their counterparts in geometrical space. This is a concept that makes sense from an artistic perspective as well: if a face is bigger, it should get more texture detail (a larger UV space coverage) than a smaller one.

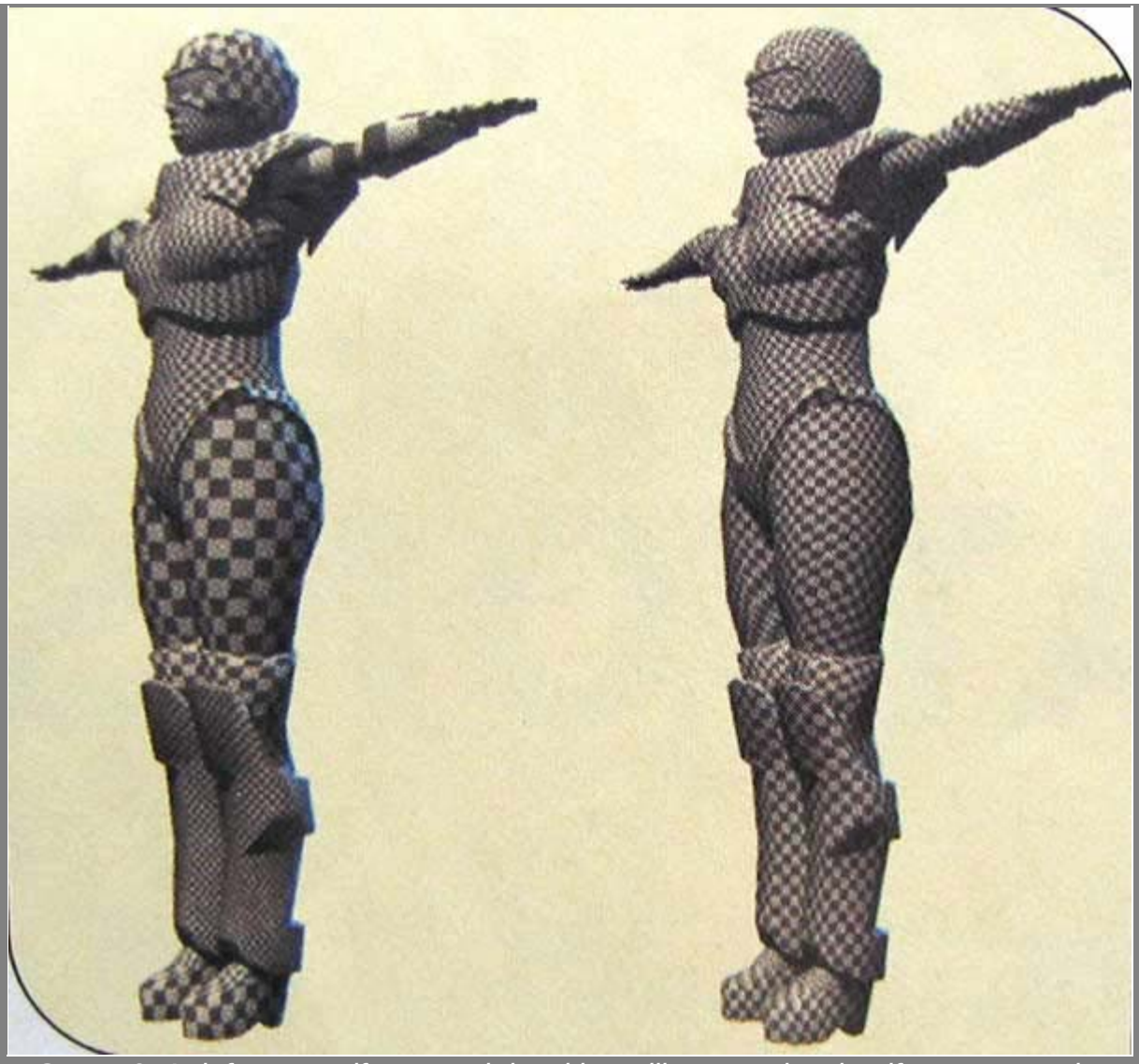


***FIGURE 6.*** *At left, non-uniform texel densities will create visual artifacts on certain platforms. At right, the texel density is uniform as a function of geometric space. If this was a fill-bound mesh, the unequal texel density would also cause cache misses to occur on certain platforms, effectively increasing the total fill cost of the mesh.*

The concept extends to objects too: if an object is smaller, it's likely to be smaller on-screen as well, and should get a smaller (less detailed) texture.

## Fill-Bound Surfaces Conquered

Following is a list of things to do and look out for when constructing fill-bound geometry:

- Build mip-maps for all textures.
- Shy away from large surfaces with complex material properties, such as bump maps and glossy materials.
- Don't overlay several very large transparent or additive layers.
- Don't make large wall/ceiling segments double-sided unless you absolutely must. If your engine doesn't support back-face culling, make sure to get rid of large, unnecessary back faces.
- Choose your texture formats intelligently to save texture space. If you do not

have access to compression formats such as DXT, see if you can't palletize textures.
- Use small texture swatches or diffuse materials instead of larger textures, even at the expense of vertex splits.
- Tweak your UV maps to distribute your texel density as uniformly as possible across the surface.

The good news about fill-bound surfaces is that, although adding more vertices probably won't help, it probably won't make much of an impact until your vertex density is high enough for your mesh to become transform-bound. (However, very large polygons can on some systems trash the texture cache, effectively increasing fill time. In such cases, tessellating the polygons will actually help.)

## Be Fruitful and Optimize

If your head is spinning by now, remember Douglas Adams's motto: Don't panic. Although there is a lot more to performance-friendly content than meets the eye, building efficient content can become an intuitive, natural process with practice.

Whether they are vertices, texels, objects, or textures, it's more about uniformly distributing them than about plucking out detail. This is a very powerfully intuitive concept: things that are smaller on-screen should get less detail than things that are bigger on screen.

Programmers can always optimize their code to go just a little bit faster. But there's a hardware limit they can never cross without sacrificing visual quality. If you are pushing the limits of your system, chances are that it is your content -- not code -- that drives the frame rate in your game.

***About the illustration***
*The mesh of the character at the beginning of this article, consisting of 2283 vertices after mesh conversion for in-game readiness), was constructed with real-time constraints in mind and makes use of a lot of pointers discussed in this article: texture seam reductions, uniform texel density, minimal material changes, smoothing group optimizations, and others. Image created by Danny Oros.*

***Guillaume Provost***
*Originally hired at the age of 17 as a lowly systems programmer writing BIOS kernels for banks, Guillaume has been trying to redeem himself ever since. He now works as a 3D graphics programmer at Pseudo Interactive and sits on his local Toronto IGDA advisory board. You can contact him at depth (at) keops (dot) com.*

**Beautiful, Yet Friendly Part 1: Stop Hitting the Bottleneck**
**Beautiful, Yet Friendly Part 2: Maximizing Efficiency**
*Articles originally published in Game Developer magazine, June and July 2003*