1. Introduction

First of all my goal is not to show particular people that they're doing things wrong. If that's what I wanted I would be probably arguing at forums :)
My goal is to help people do things better and I'm hoping this little document will be usefull.

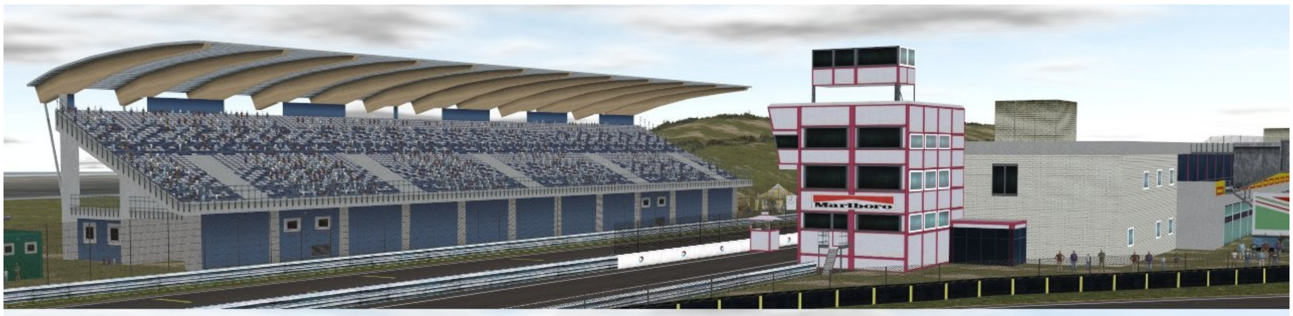
Simply put – if I want things to get better, I better stop talking and start helping :)

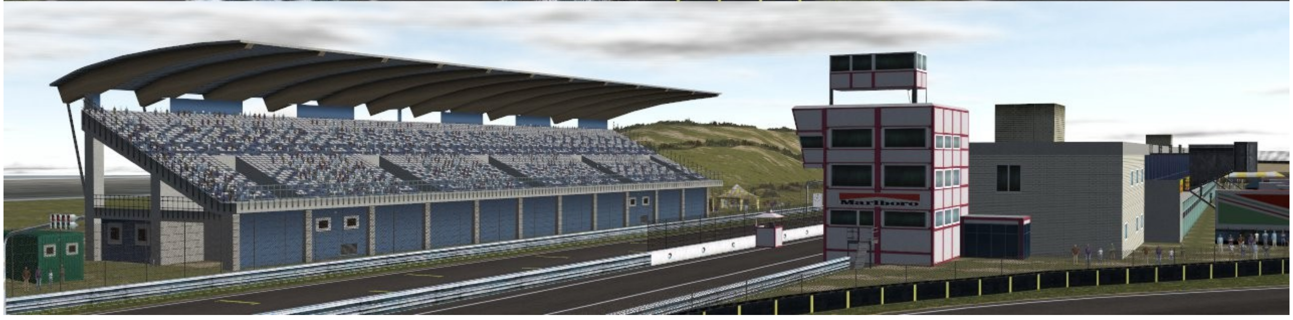
Now why I dare to say people need to read this?

Well, I'm a league admin, but some of you may also know me for my work on Shader Pack for rFactor.
While running our league we do what most admins do – we look over internet for tracks we need for our calendar and we adapt them to our needs. Usually it means fixing some issues with flying starts, pitlane entries and such.

We also update tracks visually and I have to say some tracks we find are of very poor quality. This goes mostly for conversions, but sometimes even content made from scratch, although being nicely modelled, has principal errors.

Let's start with this little comparison. We're going to have race at Zandvoort and here's a track I've found:





Upper image shows original models – look at buildings and grandstands – they're equally lit from all directions. In other words – lighting is not working at all on this track.
Bottom image shows what I did after just a little extra work – and it was just the beginning.

I really wish people would spend that little extra time on making their track react properly to lighting and not looking so horribly flat like a game from Pentium MMX era.

Why won't they do it? I don't know. Perhaps some of them are too hasty to release their work, but there may be some who care about quality of their work, but simply don't have enough experience, knowledge or awareness of things important in making content for games.

I hope this document will help those unexperienced to make better content or make those hasty ones think twice before releasing and spend some extra time on getting things right.

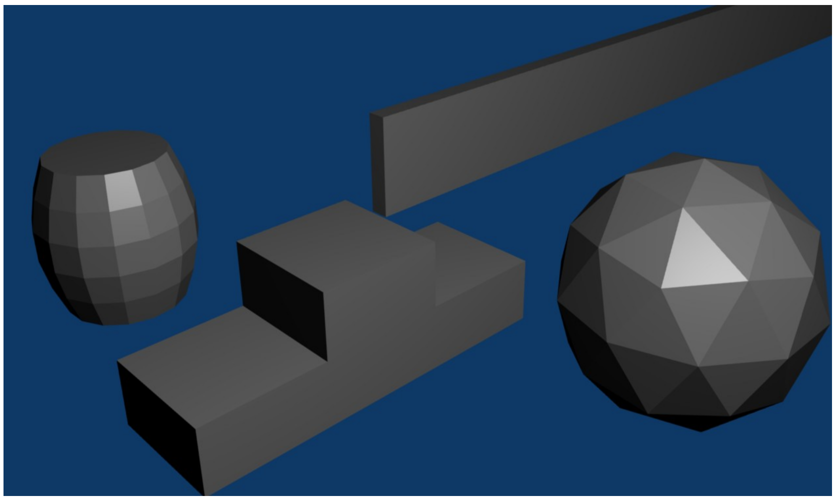
Enjoy the lecture :)

2. Geometry vs Normal vectors

We all know how it goes – everything is made up of polygons.
When we look at something made up of polygons we easily recognize the shape. If someone will model a sphere out of 50 polygons, then everyone else looking at it will see a sphere.

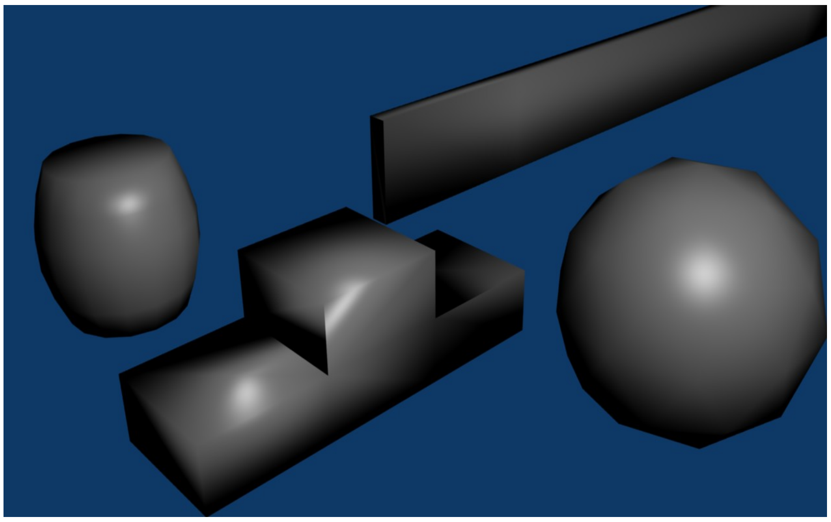
Unfortunately computers aren't that smart. Computers will see shape for what it is – a set of 50 flat polygons – nothing less, but also nothing more.

Here's an example – a barrel, a wall, a podium and a sphere:



We recognize these shapes, but obiously computer has problems giving barrel and sphere a proper look.

Let's try to give our computer a little hint and define all objects as smooth:

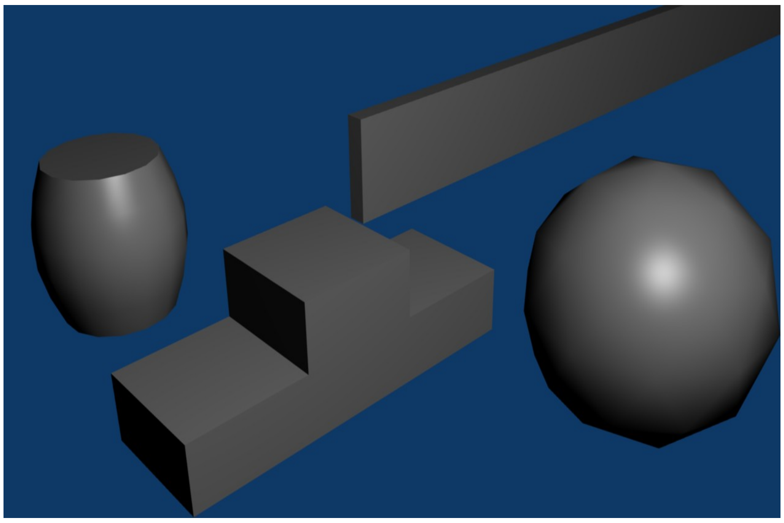


Now sphere looks fine, but barrel and wall look a bit weird and podium looks terrible.

The answer is almost obvious. Sphere should be smooth while podium and wall should not be smoothed at all.
But what about the barrel? It looked bad without smoothing and it still looks weird with it.

That's because barrel has smooth walls, but walls should not be smoothed with top and bottom of the barrel. There's an edge there and we need to keep that edge sharp.

So let's define smoothing groups for barrel. We put all polygons that make walls of the barrel into one smoothing group, while polygons making top and bottom of the barrel are put into separate smoothing group:



This time we got it right.

The conclusion:
Modelling geometry itself is only half of success. Assigning proper smoothing groups to polygons is equally important to modelling itself. Hell, it's part of the modelling!

Luckily it's much easier and quicker process than putting all those polygons in place.
Never ignore the importance of proper shape representation - give your track a chance to look good.

Well, that's it – it's not up to me to give people tutorial on using smoothing groups. There are plenty of these over internet allready. My intention was to make people aware how important that is and how objects will never look good without smoothing groups properly set.

3. Shadows vs Lighting

One of things to keep in mind when modelling is that shadow algorithms and lighting algorighms work based on different set of data:

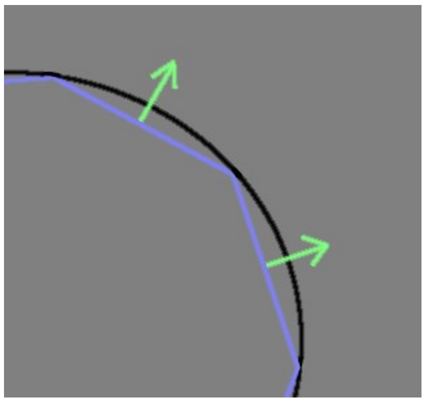
- Shadow algorithms work based on actual geometry.
- Lighting algorithms work based on normal vectors.

What are those normal vectors then?

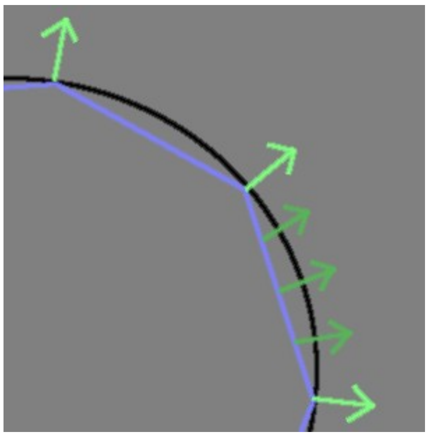
They're a set of vectors generated by your 3D modelling software based on smoothing groups you've set in your model.
They tell lighting algorithm which way a given portion of geometry is facing.

To keep it simple let's talk about two cases.

First – when your 3D model uses flat shading – you can think that every polygon has one normal vector and that vector simply points to the direction where this polygon is facing.
If this polygon is facing towards light source – it will receive direct light and ambient light. If it's facing away from light source – it will only receive ambient light.
Every pixel on that polygon will use the same normal vector, so entire polygon will receive the same level of light.



Second – when your 3D model uses smooth shading – you can think that every vertex has one normal vector.
Every pixel on that polygon will use a weighted average of normal vectors at corners of polygon.



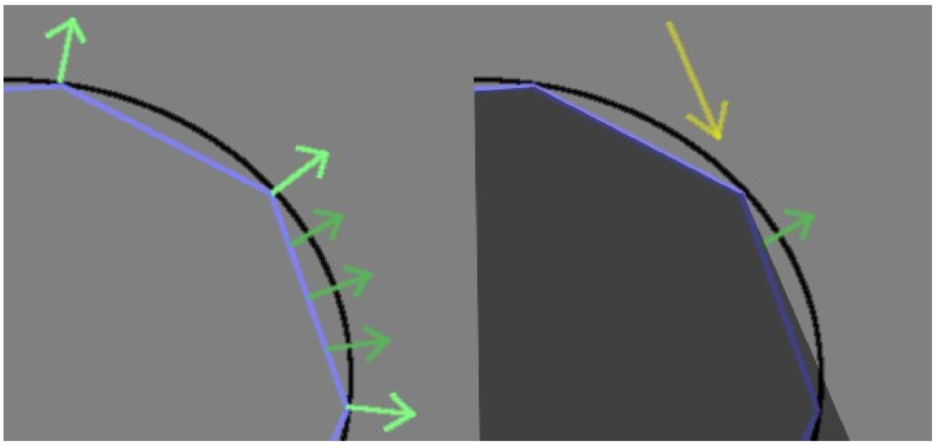
Black line represents the shape we're trying to mimic with our polygons (blue). Smooth shading is exactly what we want here, because we're trying to represent smooth, round surface.

Now what's all that talk about smoothing has to do with shadows?

Being aware of game engine limitations will help you create quality content, so let's talk about one such limitation.

Like I said – shadow algorithms work on geometry, while lighting algorighms work on normal vectors. As long as normal vectors describe actual geometry there is no problem.
Our normal vectors perfectly described shape of our polygons when we used flat shading. But we wanted our normal vectors to describe that smooth surface we we're trying to mimic with our polygons.
This is where differences begin.

Let's go back to our smooth mesh (left image). We try to describe smooth surface with limited number of polygons. Then we use normal vectors to tell lighting algorithm, that this surface is smooth.
Now lighting algorithm mimics our smooth shape nicely, but what about shadow algorithm?

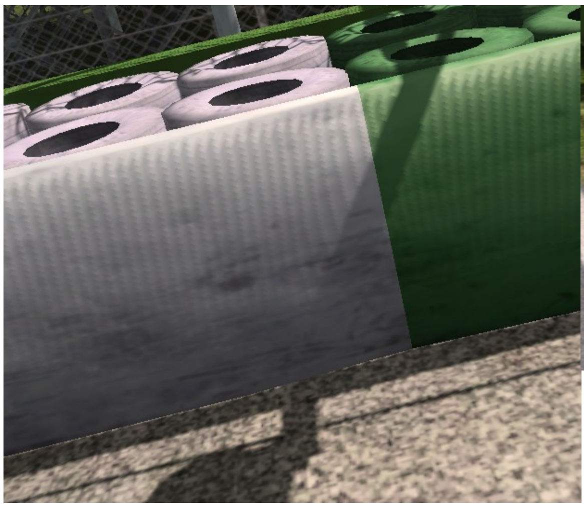
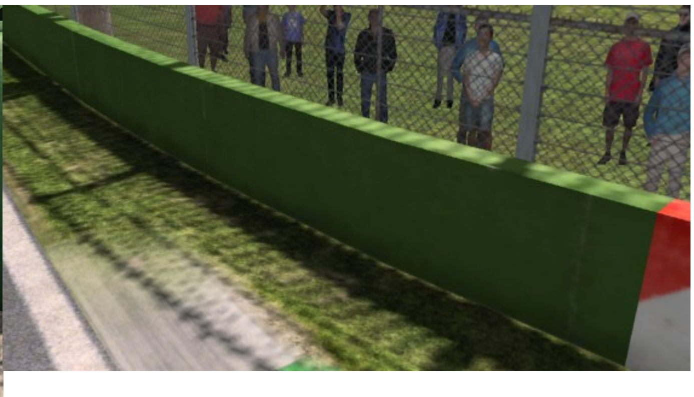


Yellow arrow on right image shows where light comes from. We can clearily see that polygon is facng slightly away from light source and therefore is in shadow. However, lighting algorithm still sees normal vectors pointing slightly towards light source on upper half of polygon.

Normally, when polygon is facing towards the light source, lighting algorithm will apply some light to it. But when shadow algorithm tells lighting algorithm, that given pixel is in shadow, no light will be applied.
The problem is, shadowmaps have limited resolution, so not all pixels on this polygon will be covered with shadow (especially the upper ones, because shadow edge is very close to polygon there).

Up to some point this is acceptable, but the bigger the difference between what geometry says and what normal vector try to mimic, the more trouble you get into.

Look at Mills – one wall has smooth vectors, the other one has sharp edges:

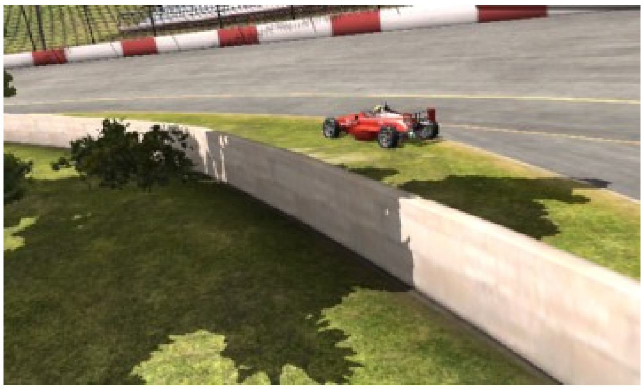
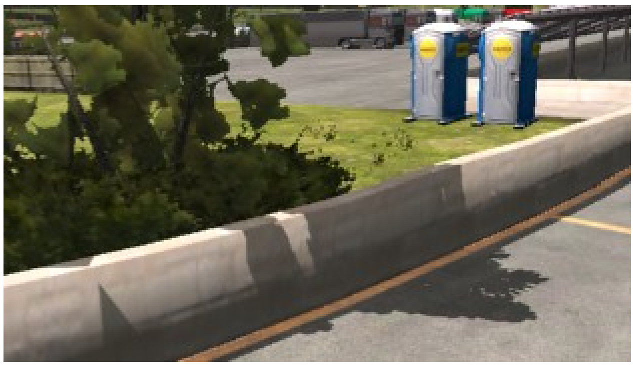
Tyre wall on left image should not use smooth normal vectors because they're actually telling lighting algorithm that it's a pipe, not a wall. Lighting algorithm is convinced that it has round shape because normal vectors at upper edge are an average of top polygons and side polygons. This means normal vectors are pointing up 45 degrees. That's a lot of difference between geometry (which say it's a box shape) and normal vectors (which say it's a round shape).

So the lighting algorithm is trying to draw a pipe, while shadow algorithm is trying to tell it, that upper part of pipe is in shadow.
We end up with black stripes that look like some kind of aliasing of shadowmap.

Wall on the right image does not try to make smooth shading between top polygons and side polygons, therefore properly describing actual shape of the wall with normal vectors.

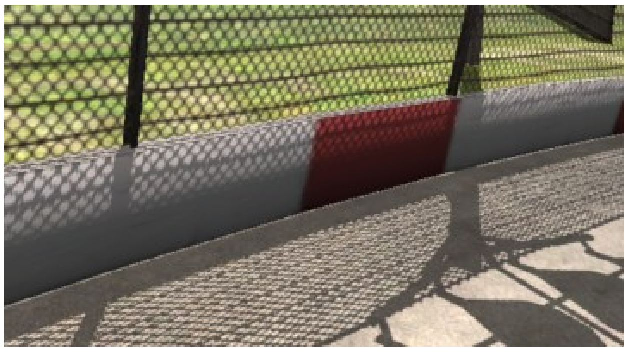
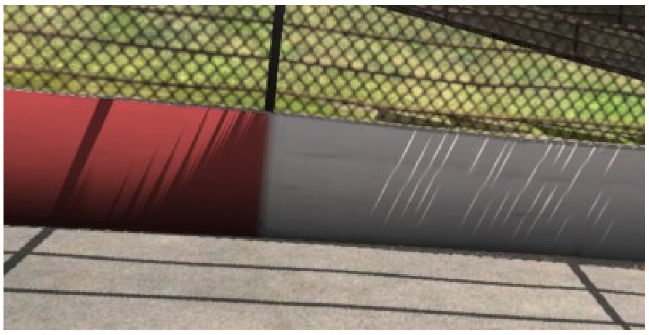
Very similar to our barrel example, eh?
But this time lighting combined with shadows made it look even weirder.

Joesville track has even more dramatic examples of that, because it has fully smoothed walls.

Both images show the same place – wall has shadows on both sides.

Same goes for outside wall:

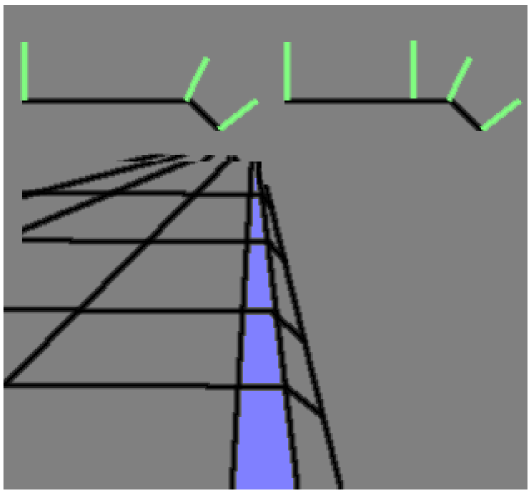



While light is on the outside (look at fence shadow on road) inner side of the wall receives light (mostly upper side of the wall). Yeah, lighting algorithm is trying to draw another pipe.

It's not just about the walls.

The most sensitive surface, when comes to errors with normal vectors, is the road surface.

One of most painfull mistakes is to have road edges modelled with some slope and then not separating these edges from main road surface with extra stripe of polygons. This causes entire road surface's normal vectors go to bust.

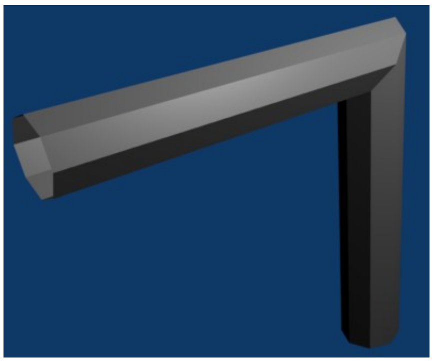


That additional stripe of polygons, marked in blue will protect your road's surface normal vectors from being influenced by edge polygons.

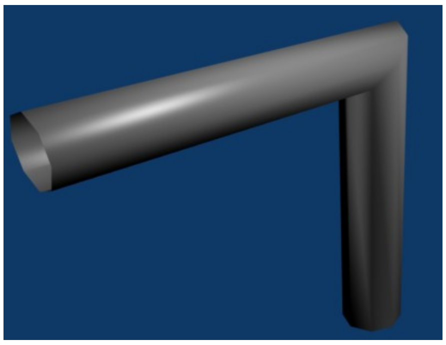
As long as track has poor lighting or no lighting at all, no one will notice these errors. Problems start when you try to use more advanced shading. All algorithms will go crazy, when fed bad data coming from track geometry.

Note that this also applies to walls – if you want a rounded edge, then you should „protect” rest of the wall with such additional stripe of geometry.

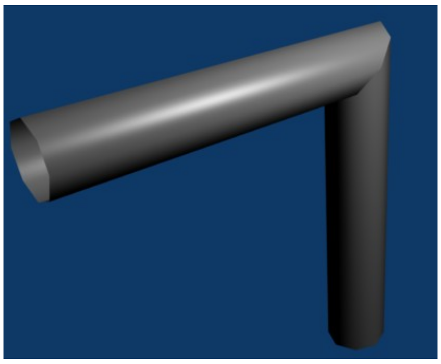
Another good example are barriers and pipes. Here's a flat shaded model:



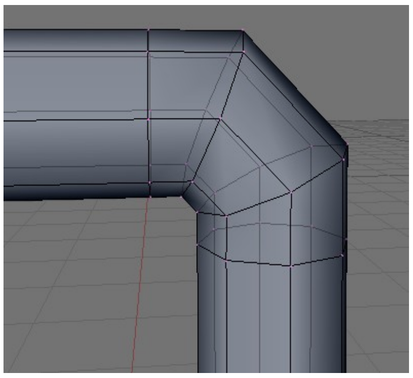
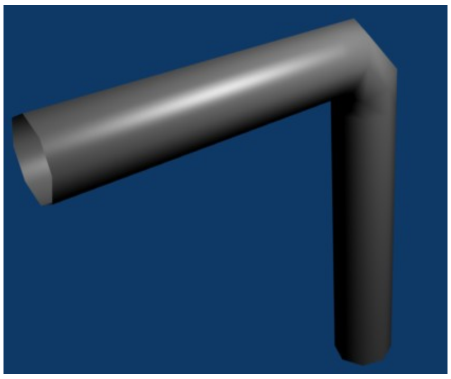
Now smoothly shaded:



It looks weird, because we have smoothing across the corner aswell. Let's try separate smoothing groups:



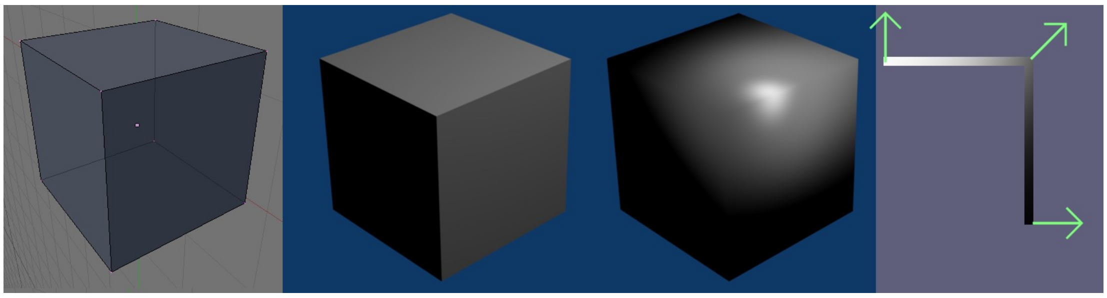
Looks good, except for the corner. So finally we do this:

Corner still looks imperfect, but it's understandable – tesselation is low. What is important, is that edges, where geometry ich changing direction are now separated from the rest of geometry, so our smoothing will remain at corner and will not propagate to straight part of pipe.

In general, more tesselation helps, but smart tesselation helps even better.

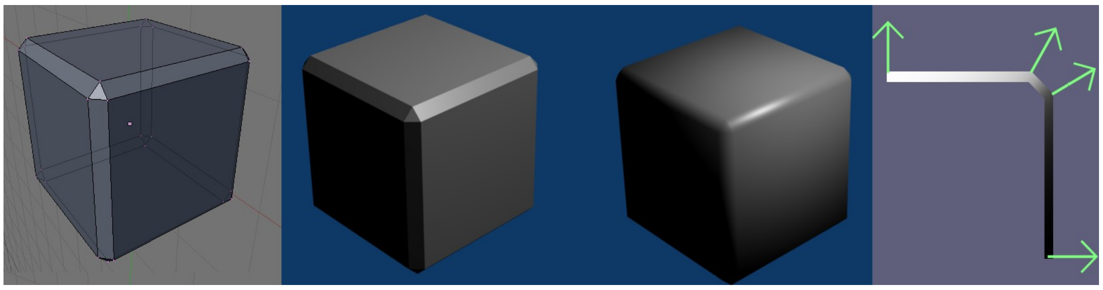
Let's say we want cube with round edges. We can't just set cube as smooth, because this will happen:



When smoothed, each vertex of this cube corners have their normal vector set to represent average angle of neighbour polygons. That's why normal vector at corner is at 45 degrees.
Lighting coming from this vertex is smoothly interpolated across polygons, so polygons look round. But we only wanted edges to be round.
Note that scheme on the right side only shows one corner. Imagine that these 2 vectors at beginning and end ale also at 45 degrees, because they're in opossite corners.

Not good at all, but it's not a surprize – we haven't modelled any edges at all, so how can our game engine make round edges?

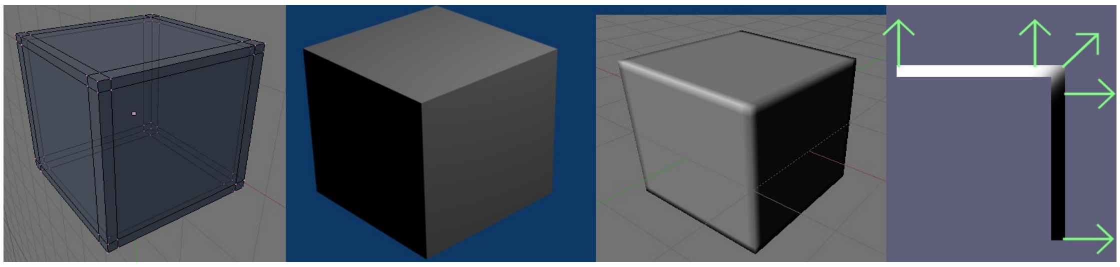
Ok, so let's bevel our edge. Now our edge it looks a lot better after smoothing, but faces of cube are still a bit rounded.



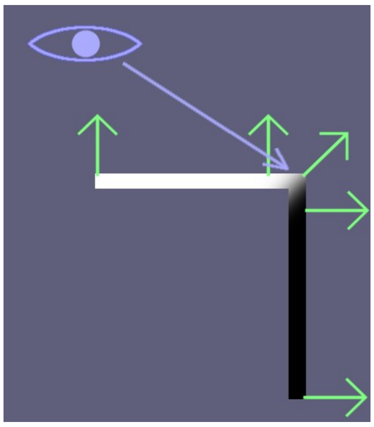
This time we have two edges so, normal vectors at edges are at 22,5 and 67,5 degrees – an average between cube face (0 and 90 degrees) and edge face (45 degrees). This means we're still telling our game engine, that cube face is rounded 22,5 degrees in each direction.

So maybe we could put our beveled edge in separate smoothing group?
No, that wouldn't help – note that edge is actually flat. The reason it looks smoothed is because it's cnnected to cube's faces. If we break that connection by creating separate smoothing group, edge will look detached from faces.

Ok, so maybe instead of beveling the edge we could use that extra strip to separate edge from face:

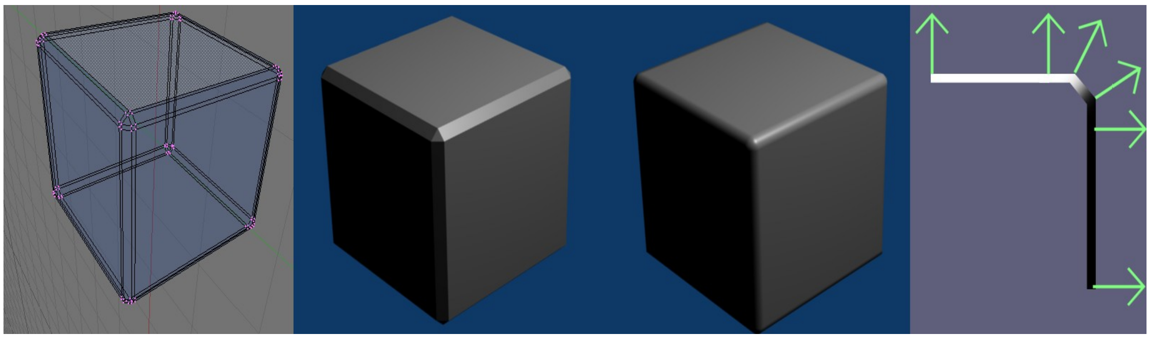


Ok, so now we get 3 vectors. Two of them are at 0 and 90 degrees and middle one is at 45 degrees. Now our faces are nicely separated from that 45 degrees normal vector at corner.
Looks almost good, but there's something wrong with edges at the other side of cube. They appear dark.
That's because our edge is now made up of two polygons – one at 0 and the other ar 90 degrees. This means we can always see half of the edge when looking from the other side – always up to 45 degrees point.



This means the edge is allready getting darker, but we can still see it.

So let's try beveling our edge again:

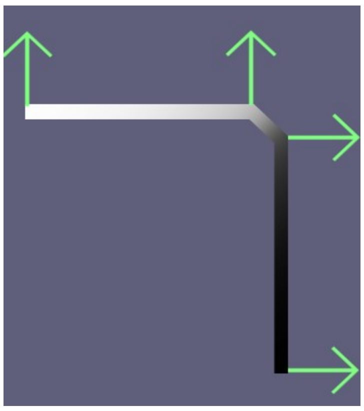


Now it finally looks good. Edge is smooth, polygons are flat, and edges away from us are facing away from us.

So we're happy now.... or maybe not?

We ended up with 96 vertices and 98 faces... Or 188 faces if we count it as triangles. That's a lot for just one cube.
Simple flat cube has only 24 vertices and 12 triangles (flat shaded faces do not share vertices and therefore each one of them has it's own 4 vertices, that's why there are 24 vertices, not just 8).

The best way seems to be making normal vectors follow biggest face.



This way we're not adding more polygons than we need and our normal vectors represent desired shape nicely.

Unfortunately both 3D Studio and Blender use simple average between two faces to produce normal vectors. I would be surprized if ISI's GMT exporter would do any better.
An option to calculate normals using face's field as weight is what we need. I believe some more professional software can do that.

With 3D Studio you can also edit normal vectors manually, but that would be an overkill.

Maybe there's a script / plugin for 3D Studio somewhere on the net that does that – I don't know. There isn't one for Blender, since blender recalculates normal vectors by itself.

So, to sum it up:

- flat shading produces no issues with lighting nor shadows, but it makes us see all the polygons
- smooth shading looks nice but causes a lot of problems with lighting and shadows if normal vectors do not describe actual shape properly – smart tesselation helps
- unweighted average used by our software to calculate normals forces us to do some extra tesselation in order to represent our shape properly
- if you make sharp edges you will have no problems, but if you want smooth edges you will need some careful work. Trying to get smooth edges cheaply, by just making a mesh smooth will ruin your lighting on areas directly connected to these edges.

There is one more solution to this.
You can prepare mesh with smooth shading + higher tesselation and use it to create normalmaps for low poly mesh that doesn't have smooth edges. This way you will have ingame mesh with lowest polycount possible and a normalmap with smooth edges on it.
Just keep in mind that normalmapping will not account for that „black further edge” problem we discussed with cube example. You may still need to bevel your edges to hide them from observer, even if you use normalmapping.

There are enough tutorials on using two meshes for normalmapping so I won't describe it here. It was definitely worth mentioning as a good technique for those who want to do more, but I also needed to give you a warning about those dark edges which some normalmapping turorials may not explain.

That's about it. Surely I haven't covered everything, but all that matters is that modders will be more aware of presence of such things like normal vectors and know how two neighbour polygons can influence these vectors and therefore entire lighting in game engine.