Modding Vision Tutorial #1: Basic "HD" Fighters

Part 1: Introduction

Welcome! This is the first in a series of tutorials that will hopefully teach you step by step how to make HD content and import it into the Vision engine for use in your own mods. The concept may seem daunting at first glance, but in reality it's much easier than it seems, especially with the advanced set of tools that the CIC community has put together over the years, it's far easier than it used to be. So if you are interested in creating new custom content for the game, or are inclined to just tinker, I hope this series of tutorials will get you started.

This tutorial will teach you how to make high-poly models for vanilla Wing Commander Prophecy: Secret Ops (WCPSO) with the Open/GL patch installed, which are easily back-ported into the original Wing Commander Prophecy (WCP). I chose WCPSO because it's free, although GOG has the original WCP Gold which includes both the original game and secret ops for less than $10 USD. Either way it's more an investment in time than money. There are many fan projects over the years, most recently Wing Commander Prophecy Unlimited which removes many of the limitations included in this tutorial, but its best if we start by mastering the basics, yes?

So let's get started!

Part 2: Getting Started – Tools of the Trade

In order to make a good high poly fighter for Vision you'll need the following tools (in no particular order):

- 3D Modelling Software – I use Blender (https://www.blender.org/download) personally and this tutorial will use it exclusively. There are other options and if you want to use them feel free. At the point you're ready to put your model into Vision you will need blender to export the file into .IFF format (more on that later) so I use the one-stop shop approach, but by no means should this restrict you.
- A copy of Wing Commander Prophecy: Secret Ops installed on your machine (and one you don't mind vivisecting). I have two separate installations so I can compare the original game to my modified copy. (http://www.wcnews.com/wcpedia/Enhanced_English_Installer_Package)
- The Blender Commander script for, obviously, blender. This allows you to import the original game .IFFs and export your work into the same. (https://github.com/Talon1024/WCPBlenderExporter)
- Some form of art software for making textures. Some examples are: Paint.net, The GIMP, Photoshop, etc.
- Treman – a program for unpacking the .tre files and allowing you to edit the game. (http://hcl.solsector.net/archive/)

- WCPascal – A program which converts the .IFF format into Pascal allowing you to edit them directly. (http://www.wcnews.com/wcpedia/WCPPascal)
- WCPEdit – a ship editor that has many uses, in this case I use it to tweak hard points, more on that later as well. (http://www.wcnews.com/wcpedia/WCPEdit_Ship_Editor)
- Texture Converter – a program that will convert your .BMPs to the .MAT format that WCP uses
- MATflags – a program that allows you to modify the .MAT files to accept specular maps. This isn't a necessity as the vanilla game doesn't support specular maps, however the Open/GL patch does so it's up to you if you want to use them. Personally I think you should, as the overall effect is far superior to a flat model.
- DOSBox – many of the tools, treman for example, predate modern 64-bit operating systems and will not run properly or at all in the command shell. For example, BMP2MAT a program that predates Texture Converter, will run in the command shell but will totally mess up the transparency of the material. DOSBox allows you to emulate a traditional DOS environment which will allow older software to run as intended. Note – I am also assuming you are using a PC running either Windows 7 or later, all my work is being done in Windows 10 64-bit for reference.

I know this seems like a lot, but you'll be doing the majority of your work in the 3D modelling and art software, the rest is just to get you started and to handle conversion into the proper game formats. If this hasn't scared you off you'll be in good shape!

Part 3: Setting up the Workspace

This will be a quick section detailing getting the game installed and ready for modding. I won't go into too much detail as most of the software has fairly robust documentation. If you have questions feel free to post them to the CIC forums under fan projects and editing and one of us will answer quickly. Remember, there are no stupid questions only stupid AI wingmen.

So if you haven't done so already, you should begin by installing WCPSO. If you're using the enhanced installer above, the default options will install all 7 episodes, the speech pack, and the Open/GL patch. Once that's done you will need to extract all the .tre data into its unpacked state. This step is a bit more complex but it's not terrible.

Next we need to make some modifications to our install of WCPSO. By default the vanilla install of WCPSO doesn't include a model viewer or mission simulator. The code is still there, but the default .ROM file (the file that creates the Cerberus' ready room) doesn't include these functions. So we need to get a new one. Lt. Overload has created some rather nice custom .ROM files on his website (http://wcsocm.weebly.com/other-files.html). Find a .ROM that strikes your fancy and follow the instructions to install it.

First, make sure you place a copy of **treman** and **CWSDPMI.exe** in the directory where you installed WCPSO. Second, fire up an instance of DOSBox, then mount the directory where you installed WCPSO as the C: drive. Third, run treman to extract the data from **data.tre**. I recommend using the –full option rather than extracting the individual directories; we will explore many of the directories

throughout this tutorial so I won't go into detail here. This can take a bit so it might be a good time to go get yourself a fresh Mountain Dew and a bowl of Cheetos, the cheese powder coating the keys is a must. A thread containing the files can be found here: (http://www.wcnews.com/chatzone/threads/treman-on-64-bit-systems.28053/)

Once treman has worked its dark magics upon your installation of WCPSO, you should see a number of new directories like MAT and GAMEFLOW. This is good stuff and we'll be getting to it in a bit. Next you will want to install WCPPascal. You may need a Borland library file VCL50.bpl in order to run the program after install, the program will alert you if you need it; if so, download the file from the web and copy it into the directory where you installed WCPPascal. That should be enough to get you going.

Finally, you should install WCPEdit. This should be fairly straight forward and not require any tweaking.

That's all you need to get WCPSO ready for modding. See, that wasn't so hard? The texture software (Texture Converter, MATflags) are stand-alone EXEs and do not need to be installed. I recommend placing them in a separate directory outside of the installed WCPSO directory so you can keep tabs on it. So for example if I've installed WCPSO into C:\games\secretops, I created a directory for them at C:\games\BMPtools. Your mileage may vary.

So to sum up:

1. Install WCPSO
2. Install a custom .rom file to enable the ship viewer and mission simulator
3. Copy Treman.exe and CWSDPMI.exe into the directory where you installed WCPSO
4. Open DOSBox, mount the directory for WCPSO as C: and run treman to extract everything from data.tre
    a. Get snacks – stay hydrated/caffeinated
5. Install WCPPascal
    a. If needed, download additional Borland Pascal library VCL50.bpl and copy it into the directory where you have WCPPascal.exe
6. Install WCPedit
7. Copy texture conversion software into a directory of your choosing.

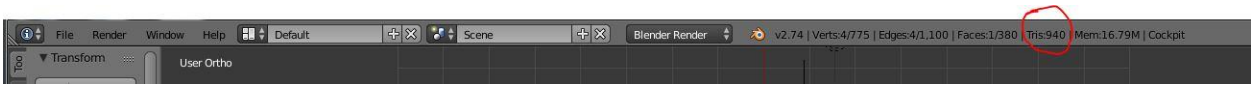
Now you're ready to start modelling!

Part 4: Building your Model

For this tutorial we'll be replacing the stock F-106-C Piranha with a new high-poly version. The original model was great for its time, but PCs have come a long way since the late 90's and we have a great many more options available to us. Unfortunately we are still limited somewhat by the constraints of the engine itself, but we can work around them using functions that exist in the engine. One of the biggest constraints we have to be aware of in working with vanilla WCPSO is that a mesh can have no more than **1,500 triangles**, not faces! Blender defaults everything to quads so be aware. You can see the number of triangles along the top of Blender's 3D object pane under "Tris". A good rule of thumb is

to keep meshes somewhere in the 1,200 triangle range. Anything closer to 1,500 can create stability problems in the engine, especially when using larger textures. Trial and error tells us that 1,200 seems to be the safest band to stay in.

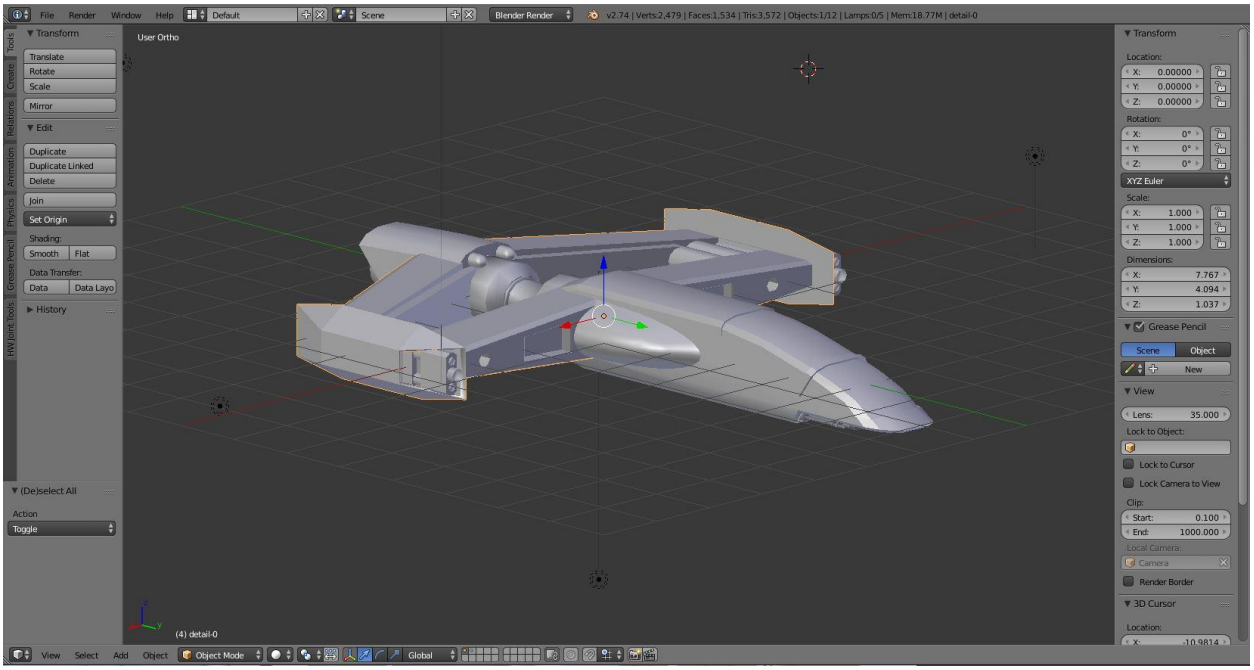
Figure 1: Triangle Counter in Blender



A note on LODs: If you import one of the original WCPSO models you will notice that there are several meshes all jumbled together, you will also see in the object tree several meshes labeled "detail-0" through "detail-2" these are called Level of Detail or LOD (sometimes LoD)s. The LODs are arranged by levels, so level 0 (detail-0), is the most detailed mesh, while the third level (detail-2) is the least. This is done so that the game engine can more efficiently govern its resources by rendering less detailed meshes at farther distances where the player isn't likely to notice. For our purposes we won't need to build any mesh other than LOD 0 since modern PCs have exponentially more horsepower than they did when WCPSO was new. However if you wish to make LOD meshes a good rule of thumb is that each successive LOD should have 50% fewer triangles than the one below it. So for example if your detail-0 mesh has 1,200 triangles, detail-1 should have around 600 and so on.

Ok enough with the boring stuff, let's make a Piranha!

Figure 2: The high-poly F-106 detail-0 mesh



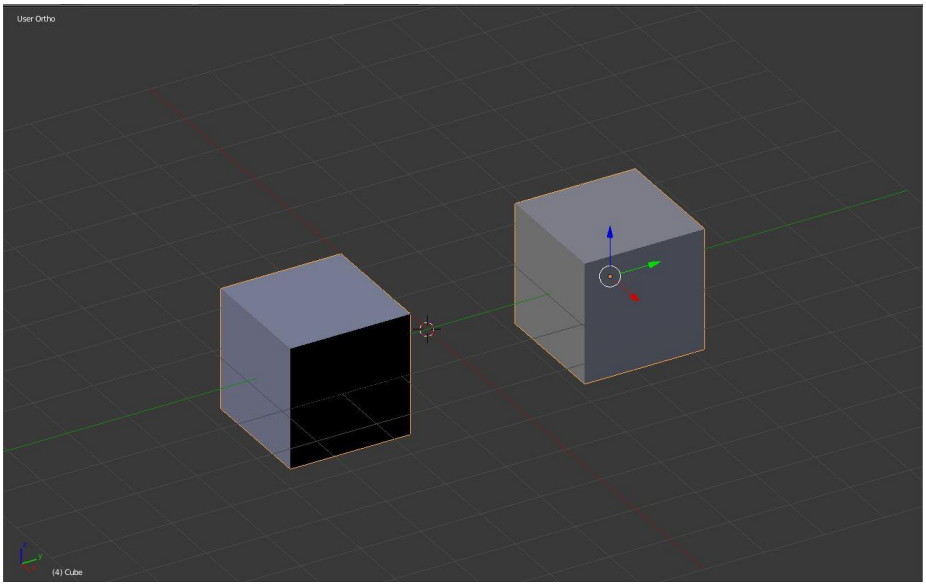
Tah Dah! Done! Ok, it's really not that easy but you didn't really expect me to walk you through the basics of 3D Modelling did you? Well I won't do that but I will give you some pointers. The first thing you might notice is that the mesh is actually 3,572 triangles much greater than the 1,500 the game

will accept. If you try and load this mesh as it is into the game it will crash, I know, I've tried. How do we work around that? We'll get to that in a minute.

First I promised some pointers so here we go. It's nice to know that the Blender Commander script is actually quite robust when it comes to handling Blender's modifiers. The most common one that most modelers use is the Mirror modifier. You can build half a mesh and blender will mirror your work along whichever axis you choose. This is particularly useful when modeling fuselages and wings because you only have to do it once and it prevents you from making scaling or cutting errors common in modeling an object manually. The Kilrathi however tend to ruin this one since their ships are decidedly asymmetrical. It's also very handy when texturing an object as you only need to do half the object and the mirrored side automatically adopts the same UV coordinates as the parent mesh. Keep in mind though that any text on the texture will appear backward on the mirrored side, so plan ahead.

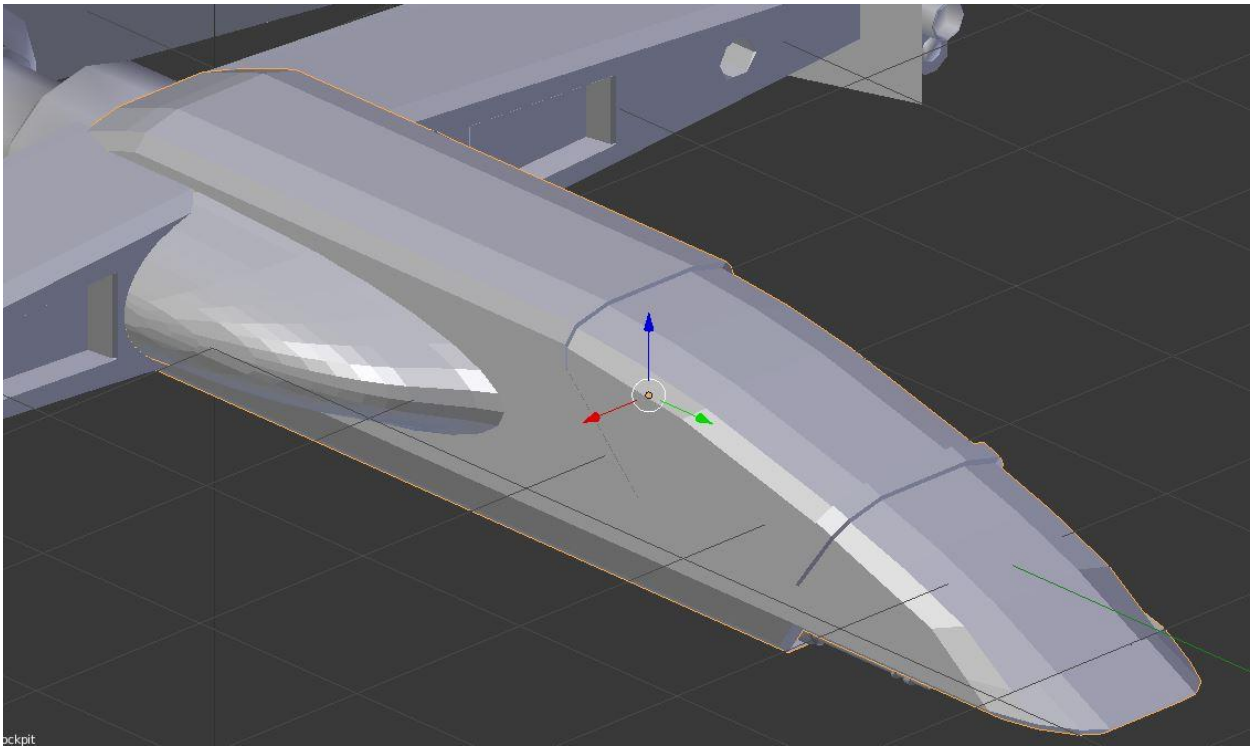
A note on duplicated objects: In cases where you need a reusable object on a single mesh and the mirror modifier won't do, like guns for example, it is common to model and texture an object once and then duplicate it (in Blender you can select an object and duplicate it by hitting Shift+D). This is fine and will cause no issues on export; however if you need to mirror a duplicated object along a specific axis (by going into edit mode, selecting all the faces of the object and then go to Object->Mirror->(whichever axis)) Blender has a nasty habit of inverting the normals of the mirrored object as well. If this happens the object will appear transparent or simply not render it in the game engine however Blender will render the object normally. In Vision, the object is still there, but the faces that display the texture are actually flipped to the inside of the mesh, so Vision won't know how to render the object and so it won't although it will still have mass, a collision sphere, etc. Fortunately Blender gives you a clue, objects with flipped normal will appear a dark gray as opposed to a lighter gray for normal-proper meshes. Fortunately the fix for this is relatively simple. Just select the mesh with the inverted normals, go to edit mode (by hitting the Tab key), select all the faces/vertexes/edges depending on what edit mode you're in (by hitting the 'A' key, you may have to do it twice if you already have a face/vertex/edge selected) and then go to Mesh->Normals->Flip Normals.

Figure 3: Duplicated and mirrored cubes. The cube on the right is normal-proper, the one on the left has inverted normals and will not render correctly in the game.



Smooth faces and smooth groups are another common tool in the modeler's toolbox. Smooth faces change how a mesh's faces receive light and shadow. By default, all mesh faces are considered "flat" meaning that the faces receive light along the planes created by the surfaces between the angles. For technical objects like armor plating, this is fine, however for beveled or rounded objects like, say, cockpit canopies or radar domes, this creates an unrealistic look:

Figure 4: the F-106 cockpit with flat shading



There are a couple ways to approach this; the first is to simply make the entire object smooth. This tends to look very bad depending on what you're working on as the edges tend to appear mushy and not sharp. The second is to use a smooth group. You can go about making a smooth group in a couple of ways, the easiest is to select the faces that you want and then select smooth under the UV/Shading menu for faces.

Figure 5: Smooth group selection

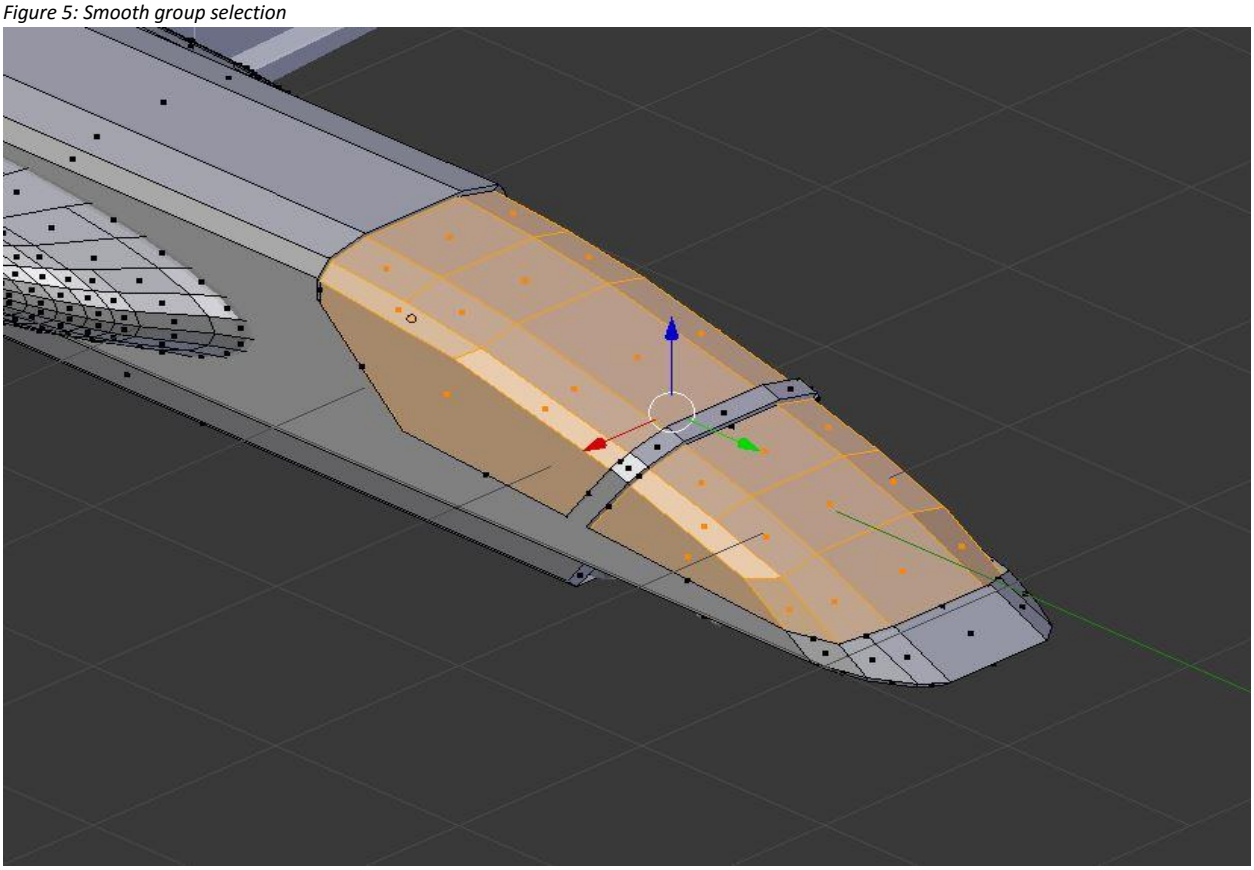
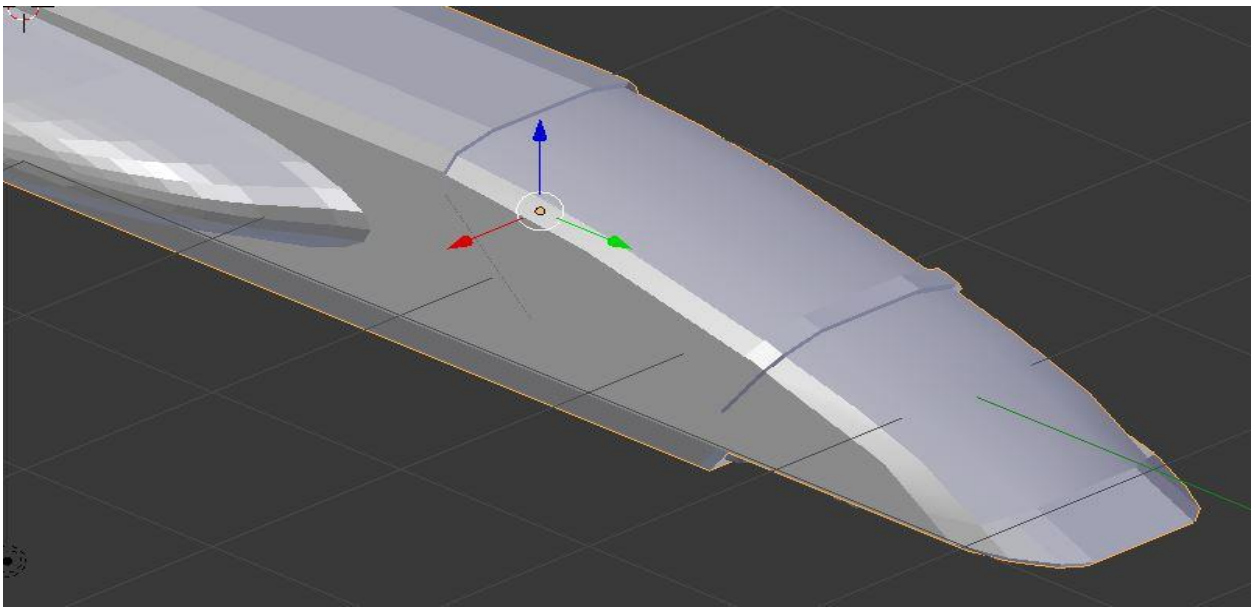


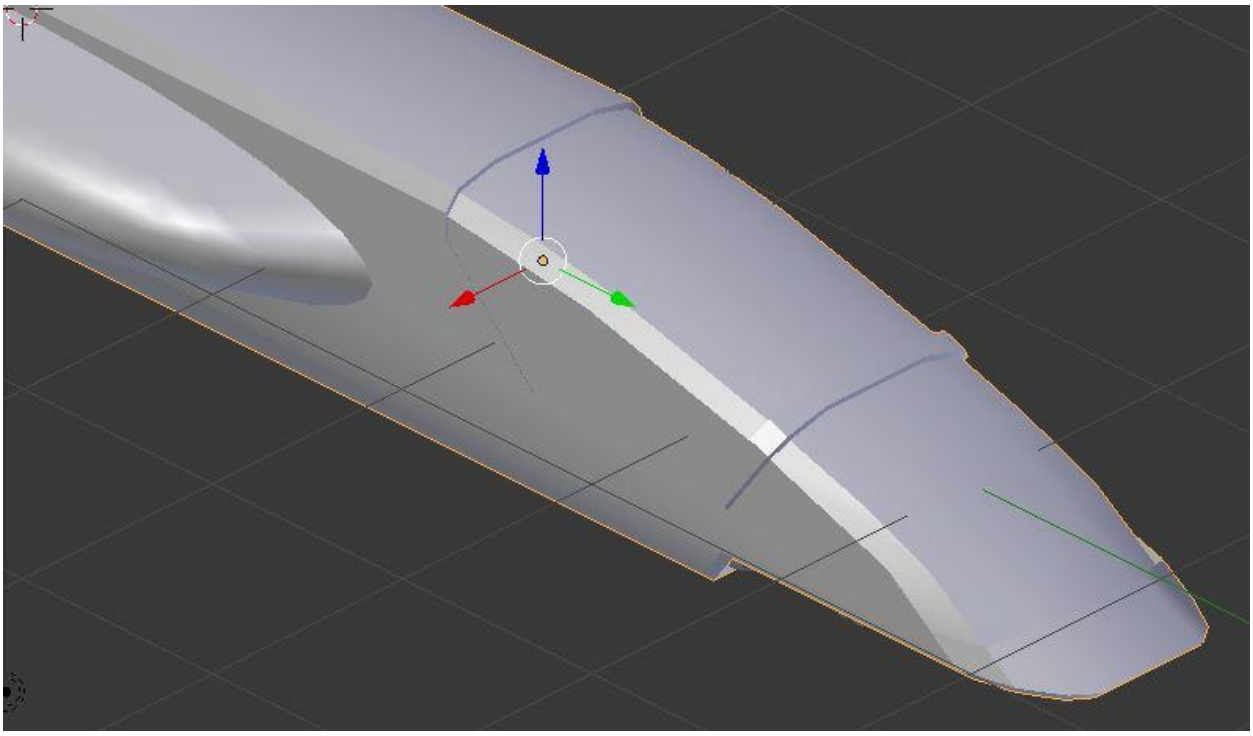
Figure 6: The canopy after smooth shading applied



Much better! However I've saved the best for last. Remember earlier when I said that smoothing the entire object would make an unrealistically shaded blob? Well the fastest way to make an object look

good is to do just that! Make the entire object smooth and then apply an Edge Split modifier to the object. You can specify at what angle you want an object to reflect flat shading vs smooth shading. The default is 30 degrees which is generally fine, but you can fine tune as you like. The end result:

Figure 7: F-106 cockpit with smooth and edge split applied

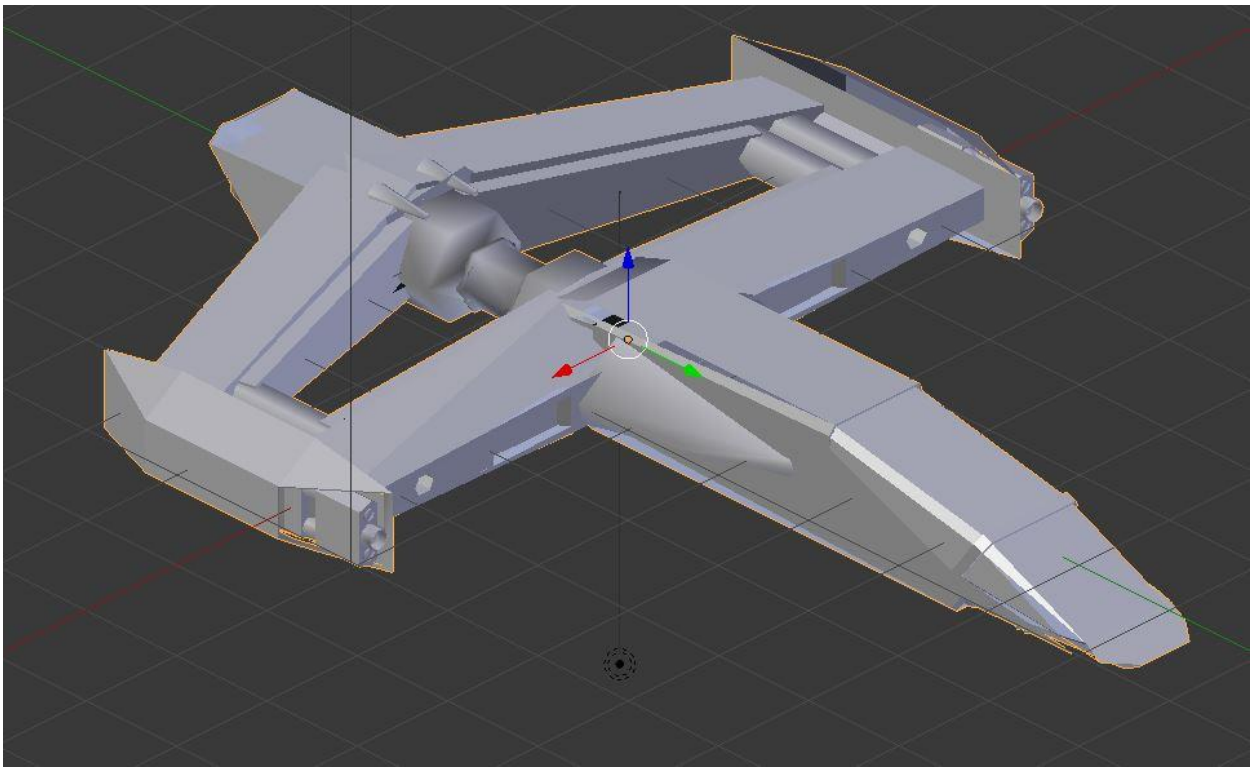


Depending on how much control you want over smoothing, this is the fastest method that gives good results.

The last modifier I'm going to go over is the Decimate modifier. Sounds sinister, right? Well it's not too scary; it's actually quite useful especially when you only have 1,200 triangles to play with. The decimate modifier uses an algorithm to collapse many faces into one where it can. There are several different types of Decimate: Simplify, UnSubsurf, and Planar. I tend to use planar most of the time varying between 1-5 degrees; the larger the degree angle, the fewer faces you have and the greater savings in triangles. However, unless the object is sufficiently simple, anything greater than 5 degrees, and sometimes less than 5 depending on complexity can cause broken edges or gaps to form in the mesh. This should be the last modifier you apply to your mesh right before you're ready to texture. Also be sure to look your decimated mesh over carefully for splits and malformed planes.

So I know what you're thinking: AH HA! So you just decimated the hell out of the mesh to get it down to 1,200 triangles! Well, you can, but then it looks like this:

Figure 8: The Piranha decimated to 43 degrees and 1,435 triangles.

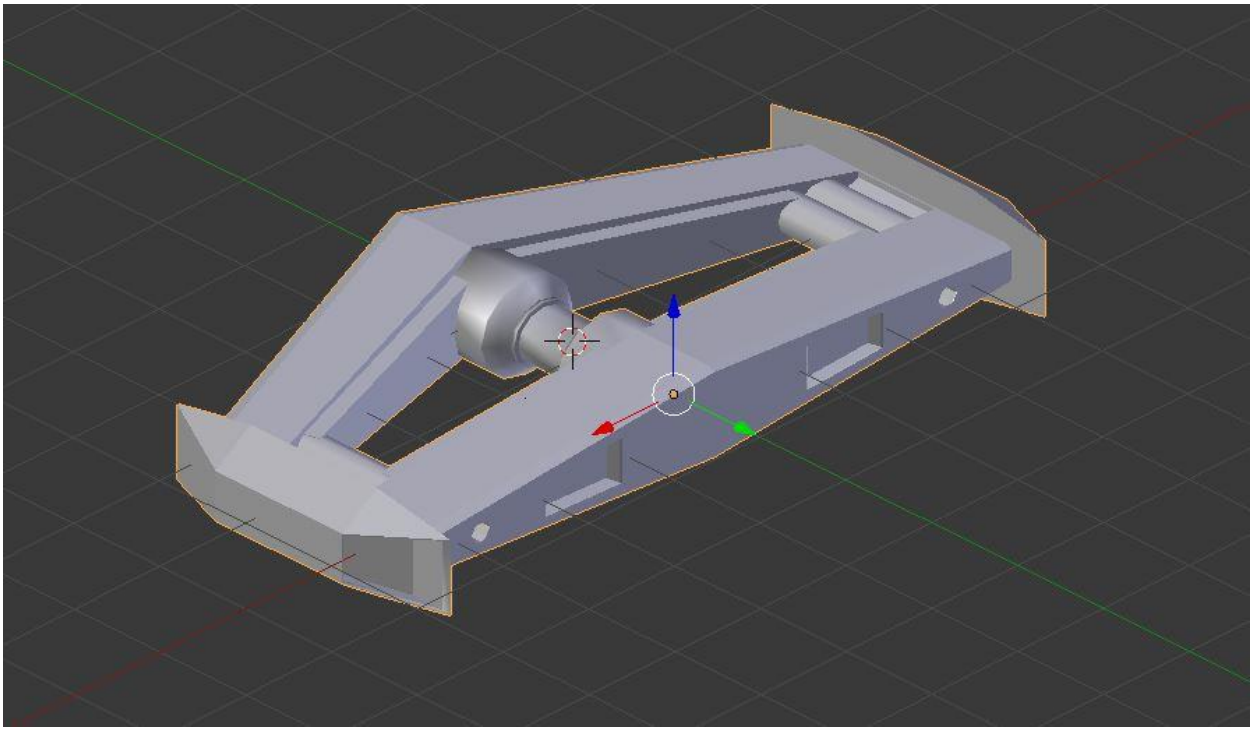


Ugh, there are no words…No what we do is to slice the mesh up into 1,200 triangle chunks and then we reassemble it later in the game code! In fact I recommend planning ahead and logically break the model up in your mind before you start, that way you can work within the bounds to make the best detailed section you can. This Piranha is actually six different meshes: the cockpit, the wings, the tail and engine, left gun, right gun, and nose guns. You have to decide first what mesh will become detail-0 and what will hang off of it as objects later in assembly. For the Piranha, I decided that the wings which have most of the mass, would be the base mesh so I named it detail-0; ideally I've found that the section closest to the center of the fighter works best. It's important that you do this, if you do not, the export script will assume an object in the scene to be detail-0 and assign it as that, but you can get very ugly results later when the script calculates the collision sphere off that object. In the game this can lead to some very odd results like having the ITTS leader and target brackets appear off center, or the mesh render in the wrong location, shooting through the mesh, etc. After you've decided on your detail-0 section, model the object around those constraints.

Figure 9: The Piranha's object tree



Figure 10: F-106 detail-0



Remember, your detail-0 object must be located at the center of the scene. The easiest way to ensure this is to first make sure that detail-0's origin is set to the center of its mass, snap the 3D cursor at the center of the scene, and then move the selected object to the 3D cursor.

A few other things to note while modeling. Parenting objects has no effect on the export of the objects to .IFF, Blender Commander doesn't support parenting. I still parent my sub-objects to detail-0 because it makes it easier to rotate and scale the object when I'm done modeling.

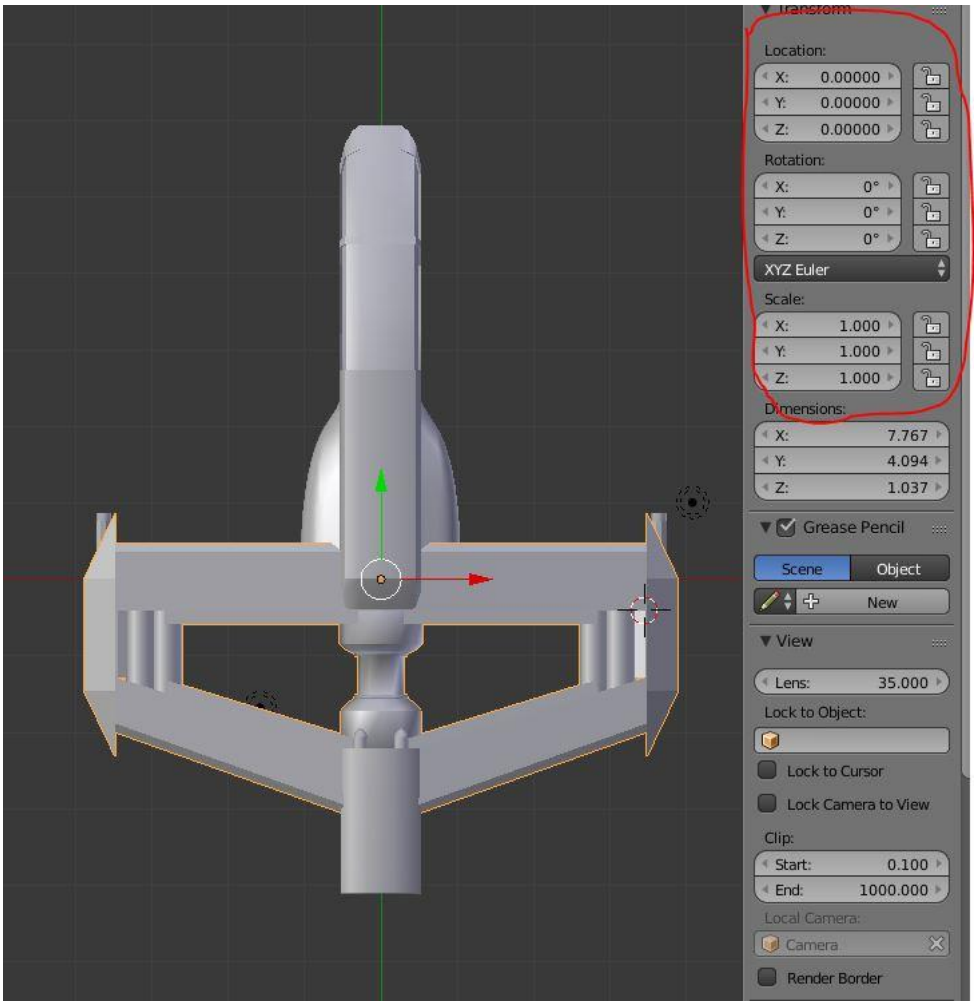
A note on scale: This is easy – one blender unit equals one meter in Vision. So if you have a 16 meter fighter, when you scale your mesh the Y-axis should read 16. When you have your mesh

positioned as you like and scaled appropriately, **you must apply location, rotation, and scale** to your detail-0 mesh **and apply rotation and scale** to your sub-meshes. **DO NOT** apply location to sub meshes, Vision uses the location coordinates of the mesh relative to the attachment hard point coordinates to properly position the sub-mesh. If you don't the mesh will scale improperly against the other models in the game (could be larger or smaller depending on the scale factor), sub-meshes can appear at wrong orientations or positions, and generally will look like crap.

A note on orientation: Blender's default orientation is a Z-up orientation. Vision, however uses the Y-axis as the vertical axis not the Z. What this means for you is that when you're ready to export your model to .IFF you need to rotate the object 180 degrees along the Z-axis prior to export. This means that if you're looking at the model from the top (keypad 7) the nose of the craft should be pointing up. Blender Commander will automatically reorient the mesh to a Y-Up orientation on export. If you don't the mesh will appear backwards in the ship viewer.

Non-Manifold meshes: As a general rule I try and avoid creating non-manifold meshes, however sometimes it's unavoidable, especially with edge split and decimate. I haven't seen any issues with non-manifold meshes in the game but it does create some extra overhead, it's best to avoid them where possible.

Figure 11: An export-ready Piranha mesh



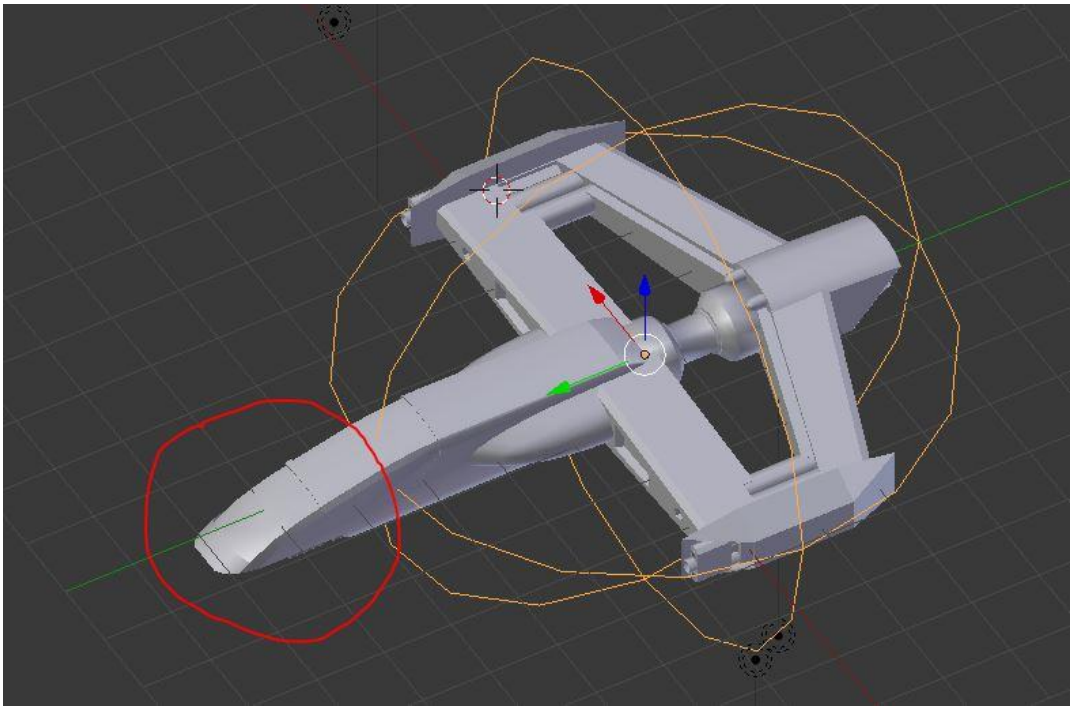
Part 5: Hard Points and Collision Spheres

Hard points are Vision's way of identifying certain points on a model. For example you need hard points at the barrels of the cannons so the engine knows where to spawn the weapon shot from; you also need hard points for missile launchers for the same reason. They also act as pivot points for rotating meshes like EPODS. For capital ships, hard points are also used to originate the smaller explosions when the ship is destroyed. You can also use hard points to attach models together.

Blender Commander uses empties to create hard points. There are two specific types of empties that are used: an empty sphere for creating collision spheres, and basic arrows for every other type of hard point. There doesn't seem to be a limit on hard points per mesh, but for fighters you shouldn't need too many. So, let's get to it, let's start with the collision sphere:

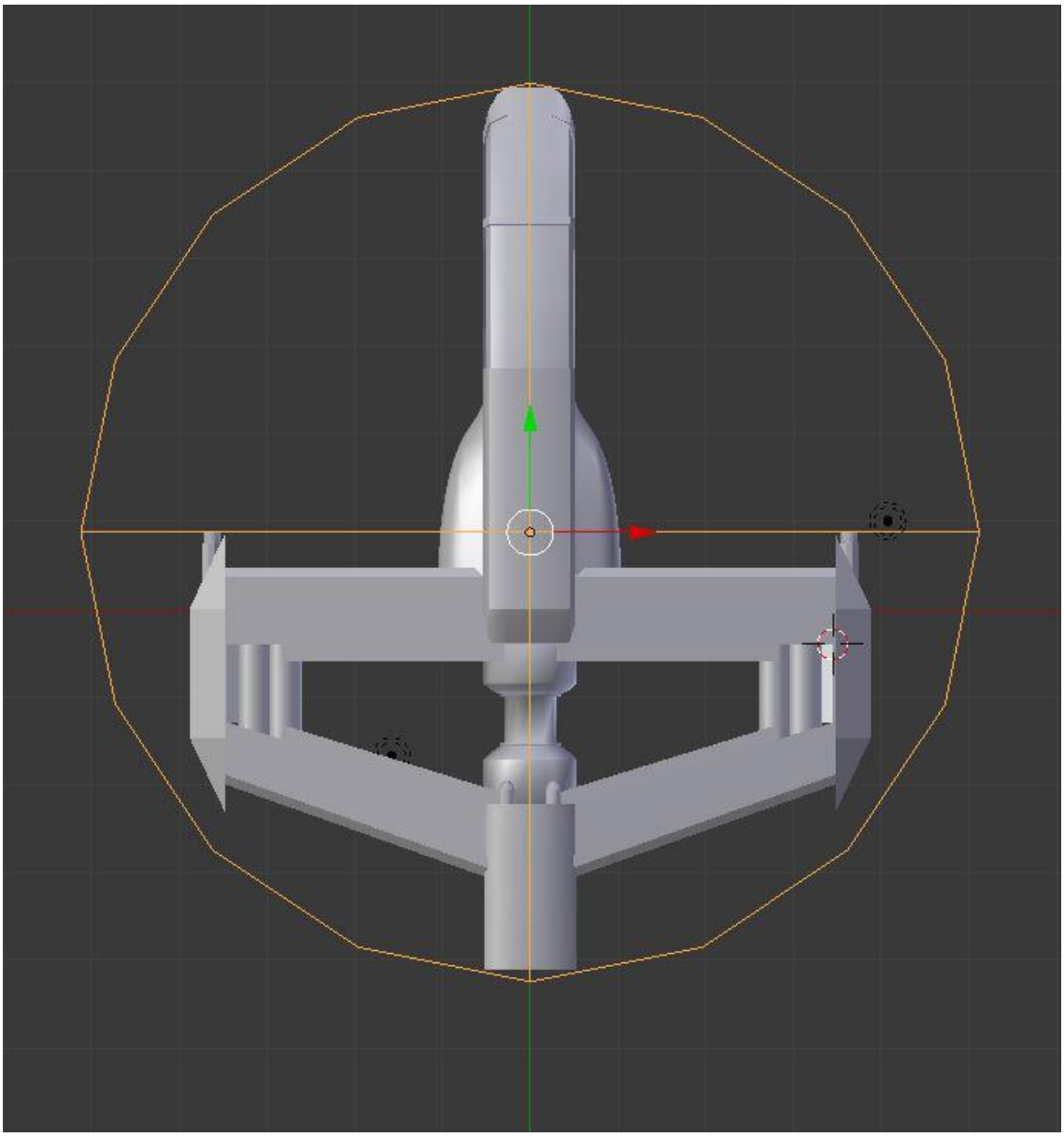
A collision sphere is pretty much what you'd expect it to mean; an empty with a coordinate radius around the mesh that Vision uses to determine if a weapon shot hits a target, if two fighters collide, and (perhaps not as intuitively) how to frame the object with targeting boxes and generate an ITTS leader. Blender Commander will generate a collision sphere automatically upon exporting an object if it doesn't detect an empty, however it will center and scale around the origin for the detail-0 mesh. In our case that would lead to shots potentially passing through the nose the Piranha.

Figure 12: Piranha with an auto-generated collision sphere



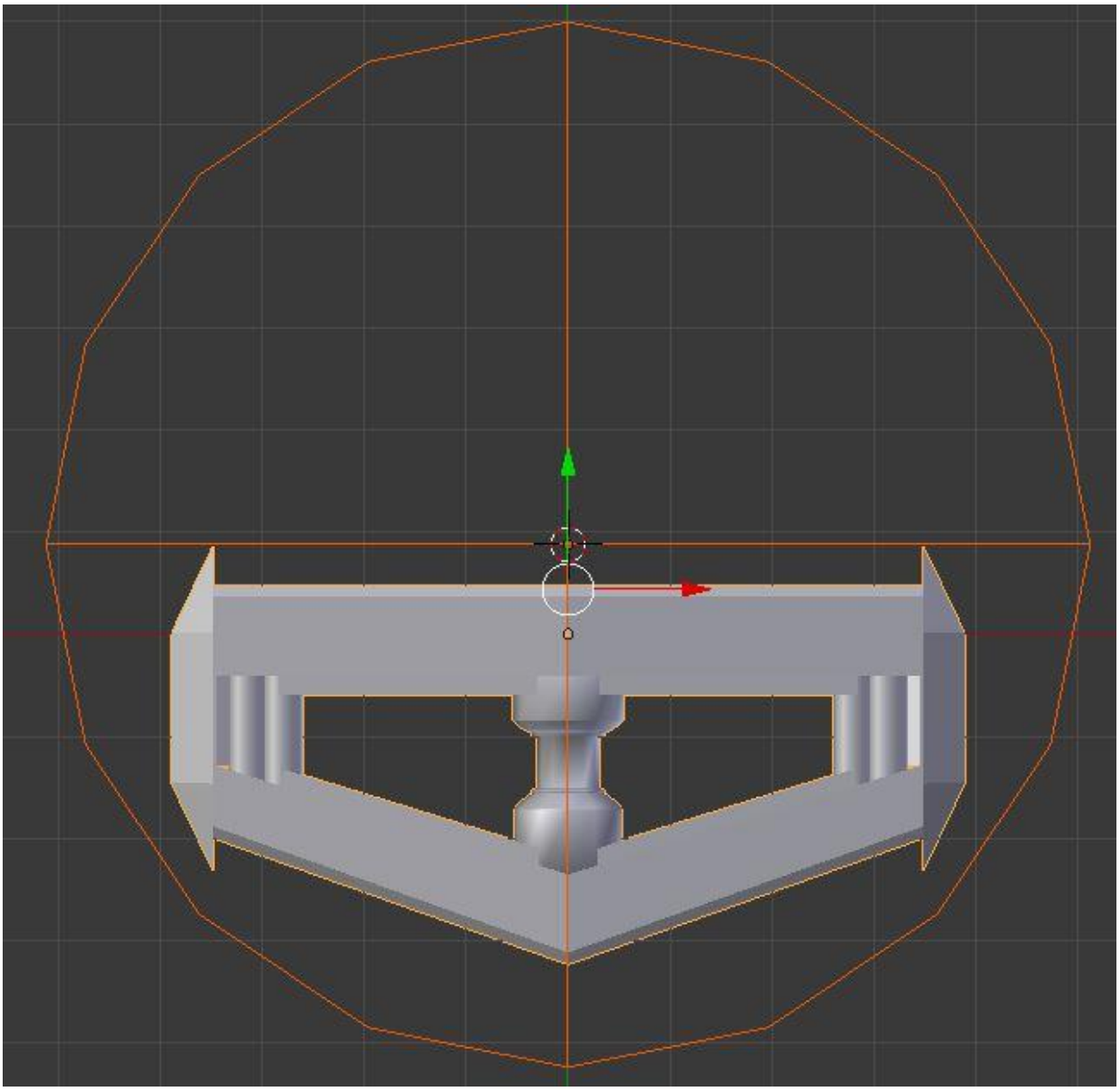
Fortunately for us, Blender Commander allows us to override the auto-generated collision sphere by creating a new empty sphere and naming it **collsphr**. You can then place the empty at the logical center of the whole fighter with the sub-meshes and scale it appropriately.

Figure 14: Piranha with a proper collision sphere



Without the sub-meshes it will look a little odd to the naked eye:

Figure 15: F-106 detail-0 and collision sphere

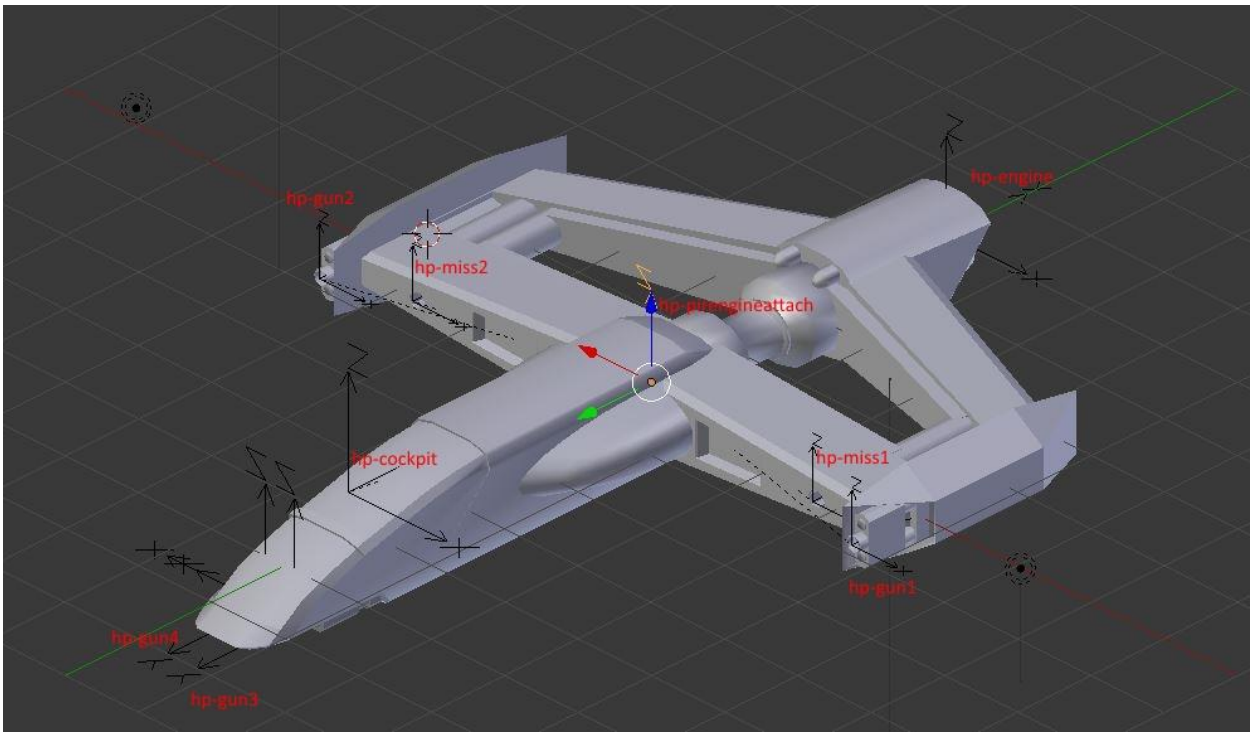


Next we want to add the hard points. The Piranha-C has 3 Ion guns, two in the wing pods and one underneath the nose; there are also two missile bays located outboard on the wings. So we need to add five hard points. Because we're replacing an existing model and we don't want to do too much re-writing to the existing ship files (more on that later), so we want to name them identically to the original model's hard points. If you're creating a brand new fighter from scratch, you can name them anything you want though I'd recommend making them easy to remember and logical to the system you are using them for. In all cases Blender Commander requires that all hard points be prefixed with "**hp-**"; after conversion the "hp-" is dropped and the text following it is included into the IFF file as the hard point name. So for example if I wanted to make a hard point for a laser gun I might name it "hp-laser1". After export, the hard point in Vision would appear as "laser1".

In looking at the original mesh, we see that there are actually four hard points for guns labeled gun1 through gun4, this is because in WCP, the Piranha had a different loadout, 2 lasers, 1 ion, and 1 stormfire cannon. Since this is a reused model from that game, the developers left gun4 in even though it isn't used. This is to our benefit actually since we want to be able to make this model backward compatible to WCP. So we add a new empty by hitting (Shift + A) go to **Empty** and select **Arrows**. This will create a new empty at the location of the 3D cursor. We then name the new object "**hp-gun1**" and position it at the barrel of the left ion gun. Repeat the process to create a second new empty, name it "**hp-gun2**" and place it at the barrel of the right ion gun. Hard points gun3 and gun4 are located under the nose and are positioned at the barrels of the ion and stormfire respectively. There are also two missile hard points, "miss1" and "miss2"; we place those in front of the missile launchers in the wings.

Note that all guns in Wing Commander cycle from left to right and outboard to inboard and grouped by type. So if we are making a new fighter with a primary armament of four particle cannons and two ion guns where the particle cannons are located in clusters of two at the wing tips, and the ion guns are in the wing roots, then hard point 1 would be the leftmost outboard particle gun, followed but the rightmost outboard particle gun, etc. The pattern is left outboard, right outboard, left inboard, and right inboard. It's not necessary to follow this pattern for your own designs, but if you want to stay in keeping with the source material, then you should. It's worth noting that the naming of the hard points don't actually influence the order in which the guns fire, that's handled in the ship file, but to make things easier on you it's best to number your hard points in the order you wish to use them.

Figure 16: Piranha with hard points, collision sphere removed for clarity



You may have noticed that there are three additional hard points that I didn't mention above; they are: cockpit, engine, and pirengineattach. Cockpit, as near as I can tell, serves no real purpose in the game. I initially thought it might have been an eye point for the player perspective, but other

models don't have it and they work fine. I left it in in case it was used in Prophecy for some reason. Engine is used as a set of coordinates to position the thruster exhaust cone mesh, and we'll get to that when we talk more about editing the ship files. Finally we have pirengineattach, this is a hard point that has to be positioned at X 0,Y 0,Z 0; this is important as we use it as a central point to attach all the sub-meshes to detail-0 in the game. When creating your own fighters using this method, always be sure to add a hard point at 0,0,0, if it's off then the sub-meshes will appear to be floating near the detail-0 mesh in game.

Ok so now we have a mesh, hard points, and a collision sphere. We're over halfway there! Now we just have to give it a coat of paint.

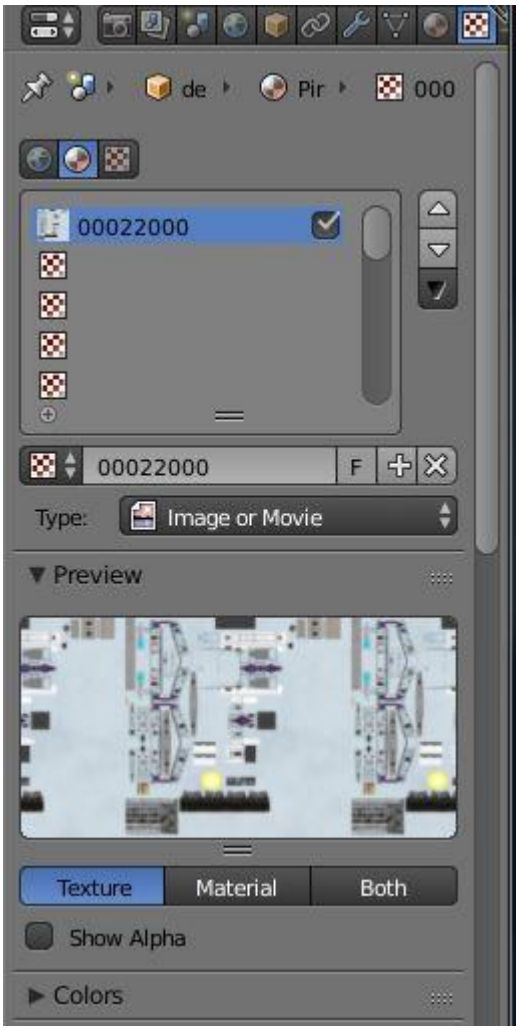
Part 5: Texturing

As with modeling I'm assuming that the reader is familiar with UV unwrapping and creating textures. I will note that in order for the export to IFF to work properly, all the faces of the model must have a texture applied to them. If you don't the file will export as a 0KB IFF and Blender will throw an error at you.

Vision has a max texture resolution of 1024 X 1024 pixels, anything larger than that will crash the game. All work must be exported as an 8-bit BMP for conversion to a .MAT file. In addition, as of this writing the Open/GL extension doesn't handle emissive or normal maps; it does however, handle specular maps. You can also modify textures with overrides for additional effects like blooms but that's for a more advanced texturing tutorial. This will focus primarily on the diffuse and specular textures.

In Blender, you will want to create a new material for the mesh. You may call it whatever you like in my case I called it "Pirhull". You only need to assign a single texture slot to the material for the diffuse texture, you don't need to apply the specular map to the model, and it will be applied later when we convert the texture from .BMP to .MAT. Next we need to name the texture; MATs in WCPSO are all named in a semi-sequential numeric format. If you name your texture "pirhull.BMP" for example, Blender Commander will nicely rename your texture for you on export to something like 00022000.BMP. It will also export a handy text file showing you the conversion. So if you have multiple textures on a single mesh, you would get a file with multiple entries. That said, the script doesn't have any kind of running memory so the next file you export will also begin at 00022000. So if you have a modification with multiple files, you need to manually override this function, it's pretty simple, just name the BMP file a number that isn't currently in use (you can find this by looking in the MAT directory in your WCPSO install and scrolling to the bottom). So for example if I wanted the Piranha to have texture 00022010.MAT I would simply name the BMP file 00022010.BMP and assign that to the material slot in Blender.

Figure 17: The texture pane for the Piranha



A couple of notes on textures. Presumably to make up for the lack of emit mapping, Vision's lighting engine, especially the direct lighting, is exceptionally bright; to the point where lightly colored faces which receive direct light get totally washed by the reflection. To counter this we need to do two things, first is to implement a specular map.

A specular map tells the rendering engine how much light is reflected from a specific part of a texture. The Open/GL patch uses grayscale data to achieve this effect, white being the most reflective and black reflecting no light at all. The quickest way to make a spec map is to take your diffuse texture and convert it to grayscale. That's usually enough to get you started but to really achieve a realistic effect you should do some more. Some tips I use are:

1. If you have a dark cockpit, gray or black for example, you will want to add some reflectivity to the texture. Glass is pretty reflective after all.
2. Try playing around with transparency settings on your paint layers, paint doesn't flatly reflect light, you might also want some of the paneling under it to reflect differently.
3. Manually lighten some panels and darken others, this creates a nice uneven effect in the lighting later.

4. Since there are no emit maps, make objects that would normally glow in your texture (formation strips, intakes, engine glows, etc.) white in your spec map. When they receive light, they will reflect as much as possible and give the illusion of glowing when combined with Open/GL's bloom effects.
5. Once you have a grayscale texture as you like it, reduce its brightness by at least 50% and increase the contrast by around 25%. Play around some. Bare metallic fighters will be naturally more reflective than a fighter that's 100% covered in paint or made of some exotic light-absorbing material.

When you're done you need to save out the specular texture as a TARGA (.TGA). The texture hack we'll do later requires that the .TGA file be named an eight digit name, but the first four characters must be 0's. I usually take the major identifying digits of my diffuse texture and use them, that way I can logically pair them together. So if my Piranha's diffuse texture is 00022000.BMP, I'll name my related specular map 00002200.TGA. Or if I have a texture named 00022001.BMP I'd name my spec map 00002201.TGA. Find a system that works best for you; however the trailing number cannot exceed 4095.

The other thing you can do to combat the over-saturation of the lighting is to fight fire with fire. You should increase the color saturation of your diffuse texture by about 50%. It will look pretty daffy in a blender render, but once you get it into the game it will look fine.

Figure 18: Piranha with over-saturated texture in Blender



Figure 19: The Piranha in WCPSO



For fighters a single 1024x1024 texture should be enough. Multiple 1024 textures would be wasteful and inefficient from a memory standpoint; in fact the original models had single 256 X 256 textures, so you have a much grander canvas to work on. When we look at massive capital ships, sometimes multiple textures are warranted, but that's for another time.

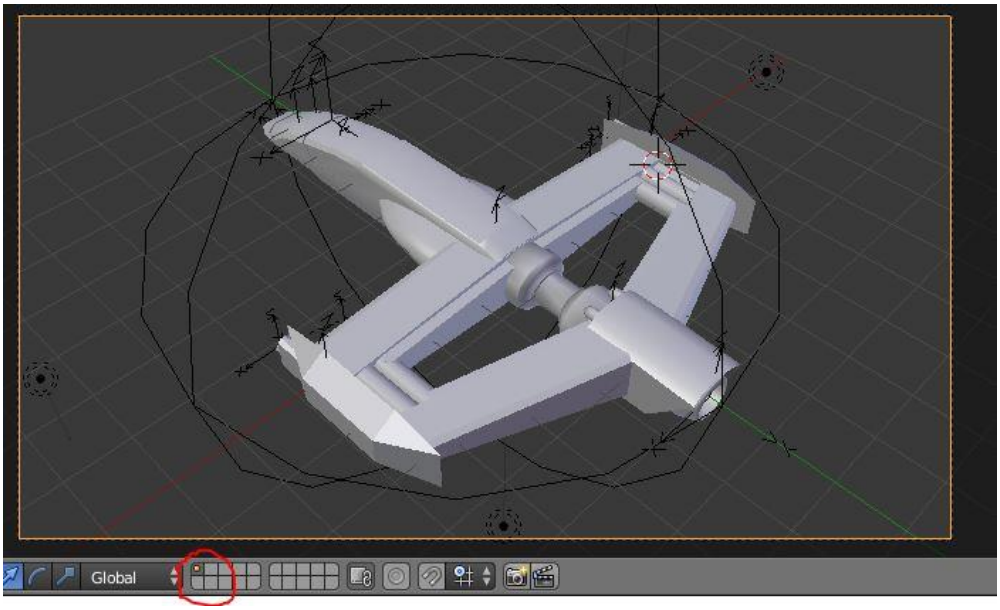
Ok that's it! You're now ready to export your work and get it into the game. Ready?

Part 6: Exporting and Converting Your Work

Well, you made it this far. Some congratulations are in order, but now comes the big payoff, where we get to catch our first glimpse of our work in the game. Fair warning, this is probably the most technical part of the process, so it will seem like a lot at first, but don't worry it's actually easier than it seems. There can sometimes be a bit of trial and error, so be forewarned, fortunately there isn't too much that can go wrong. Ok let's get started.

The first step is to export your model into .IFF format. This is the easy part, the script will export whatever object you have highlighted as long as it is on layer 1 in blender. To confirm this look at the pane under the 3D window, you should see a series of boxes about midway along the bar. They are two groups of 10. The upper left box is layer one, your mesh should be visible and the box should have a dot in it.

Figure 20: Checking Your Layers



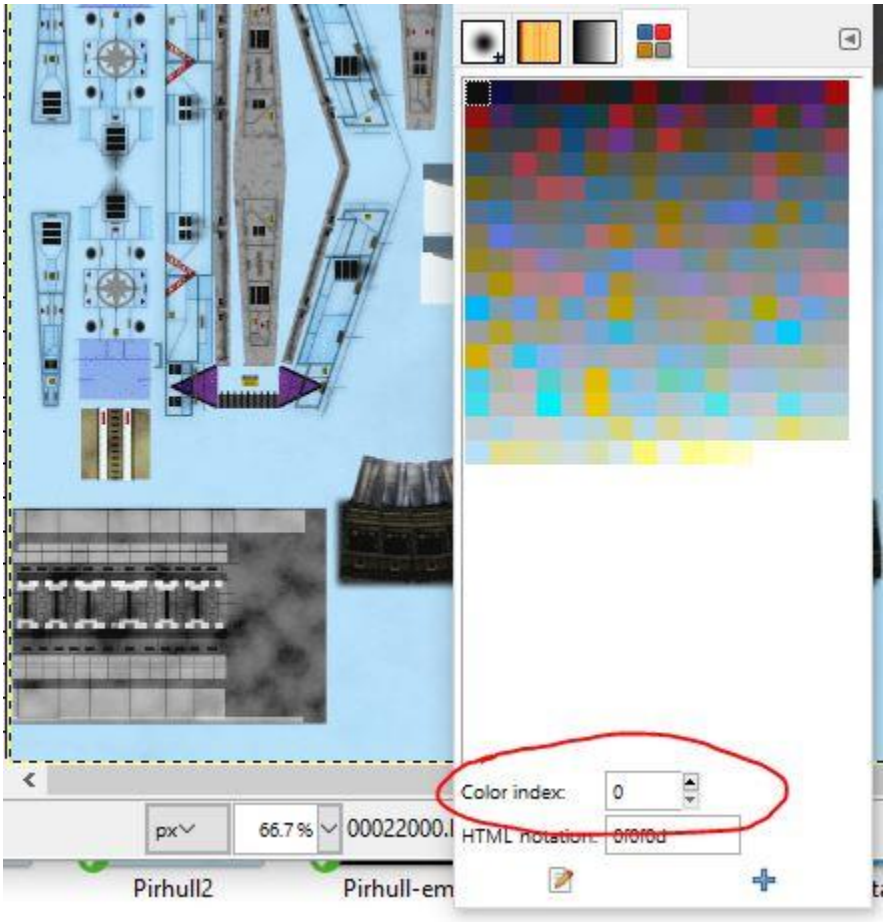
So start with your detail-0 mesh, select it, and then go to File->Export->WCP/SO IFF Mesh. If you don't see that option you should check your Blender Commander install. It's important to remember that Blender Commander will only export empties on layer one. If you've moved them to another layer and have that layer selected, it will appear as if they are on layer one but won't export. In our case, since we're replacing an existing model, I want to name the detail-0 mesh the same as the model I intend to replace, so I name it PIRANHA.IFF in the export window. As a rule, you should always back up the original models so I recommend **not** exporting from Blender directly to the WCPSO\MESH directory as that would overwrite the original file. You should then move detail-0 and all the empties to another layer. You can then proceed to export the sub meshes. We move the empties because Blender Commander would export the collision sphere and hard points for each sub mesh if we left them on layer 1. Honestly, I've forgotten to move them on a few fighters for the Model Update Pack and there are no adverse effects. As near as I can tell, the hard points and collision sphere from the base mesh overrides anything attached to it. It does increase the file size with useless data and if you need to re-import it into Blender, you have to contend with all the empties so it's best to take the time to do it right.

Once you've finished exporting the meshes to .IFF it's time to tackle the textures. Ok so you should have created two textures, the diffuse texture which should be an 8-bit BMP file, and a specular map which should be a .TGA file. You don't need to do anything else to the spec map so all you need to do is copy it into a folder in your root WCPSO install directory called **specmaps**. If this is your first spec map you may need to create the directory as it is not part of the data.tre structure. Now we have to convert the .BMP to a .MAT file for use in the game.

WCPSO uses an indexed RGB .BMP as a base format. It also interprets the color located at index 0 as transparent. Normally the default color at index 0 is true black (hex value 000000). So if you have any true black in your texture, if you convert it to a .MAT, those parts will be transparent. To avoid this

(unless you want transparency in which case you can skip this step), you can use a program like the GIMP or Photoshop to modify the color map.

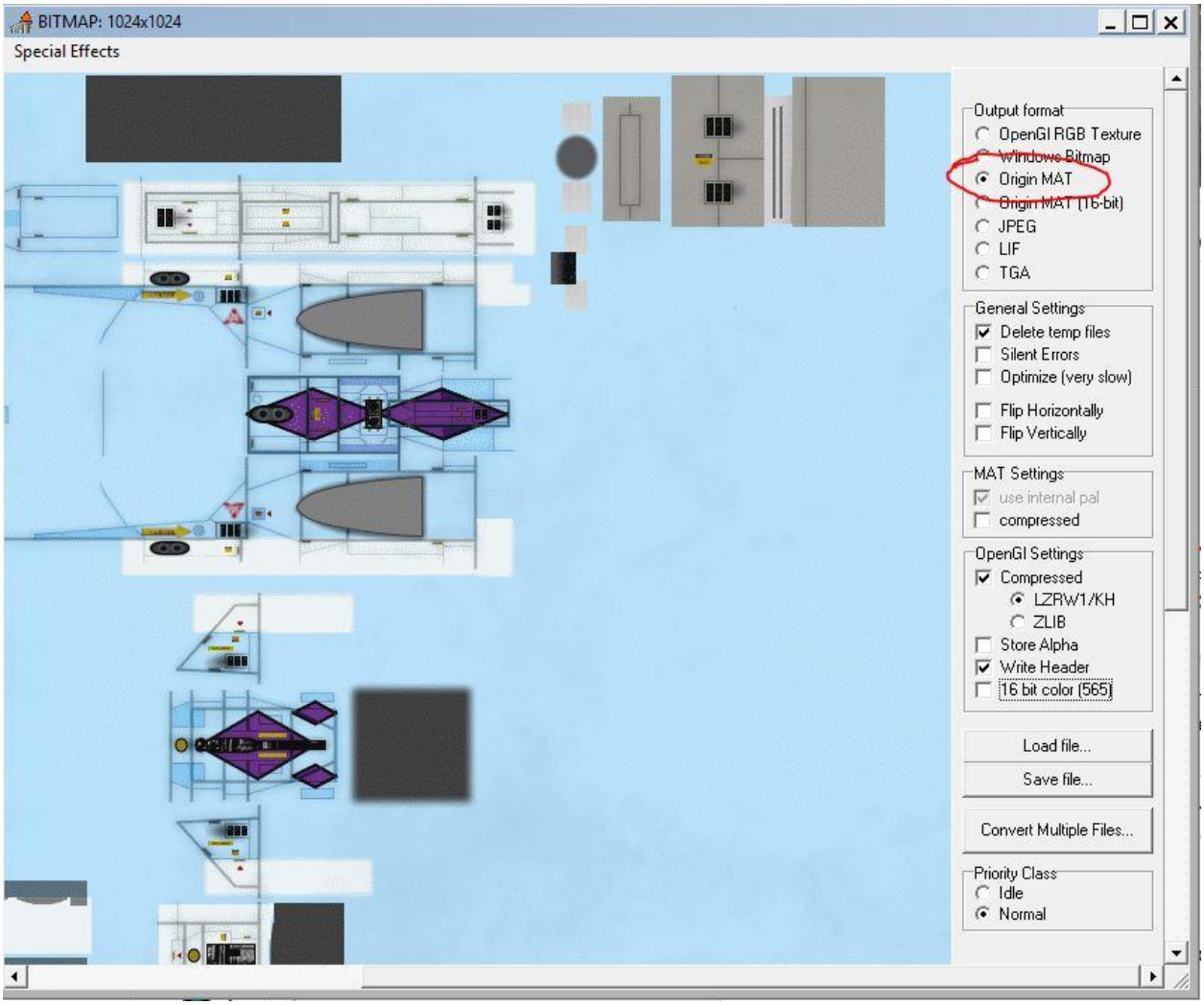
Figure 21: The Piranha's color map in the GIMP



To correct this, pick an unused color on the color map, normally around index 255 or so and if you're unsure if the index is in use, select the eyedropper tool and select parts of the texture that are similar, if the index is highlighted, then you cannot use the index. Once you find an unused color, select that index and then right-click and select "Edit Color", then select a color that wouldn't possibly appear on your texture, like a neon lime green (hex 04ff04). Alternatively, you can place a swatch of that color in an unused section of your texture. It saves some hunting and pecking if you prefer. In either case, you should move that color into index 0. You can do it in GIMP by selecting the color, right click and select "Rearrange Color Map", then simply drag the color to index 0. Save the file and now you're ready to convert it to a .MAT.

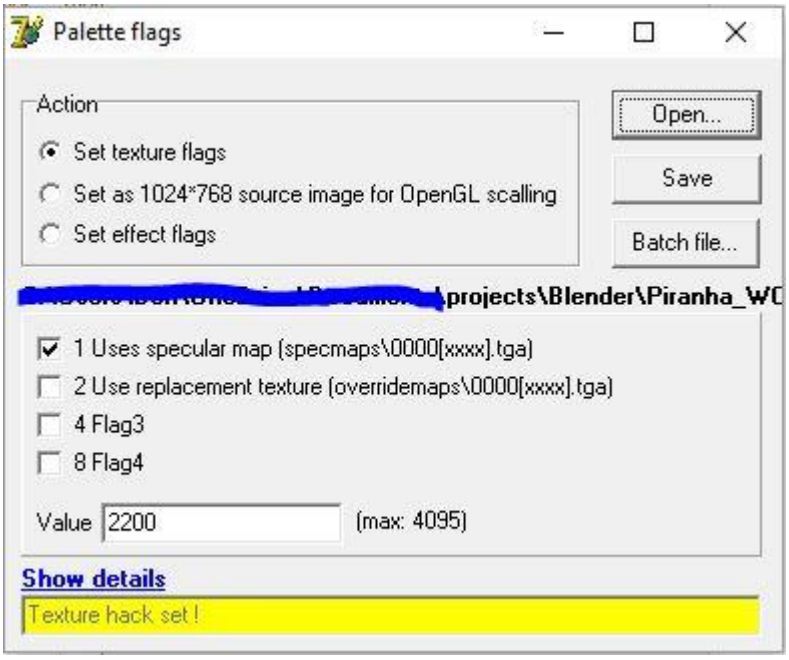
For this part just open up TEXTCONV.exe, click on the load button and open your .BMP file. Make sure you check the radio button labeled "Origin MAT" in the output format box on the upper right pane. Then click save file. Keep the file name the same as your BMP, so for example, the Piranha's BMP was 00022000.BMP so we want to save it as 00022000.MAT. Done. Easy.

Figure 22: The Texture Converter Window



Since we're using specular maps, we have to work a little magic on the .MAT file. But we have a nifty tool for that too. Open up MATflags.exe, then click the Open button and select your newly created .MAT then check the "Set Texture Flags" radio button in the action frame at the top. Then check option 1 "Uses specular map" and in the value window at the bottom, enter in **the last four digits** of your spec map's name. In our case it's 2200. Hit save. If successful the bottom window should show a message reading "Texture hack set!"

Figure 23: Successful texture hack



Finally, if you're planning on replacing an existing model, you need to back up the original mesh in the \MESH directory, in our case PIRANHA.IFF as well as the related .IFF file in \Ships. After that, copy your new meshes into \MESH and your texture into \MAT. Now you have all the pieces in place, and we're ready to put it all together.

Part 7: Final Assembly

Ok this is it, the last steps. If you're replacing an existing model you can already go ahead and check some of your work. Since we replaced the original PIRANHA.IFF in the /MESH directory with our new one, we can load up WCPSO and check our work in the model viewer. Since we haven't modified any of the ship files you should see just the detail-0 mesh in the ship viewer. It will, however let you look at your texture and specular maps and see if you want to do any tweaks. After that, we need to go modify the viewer files so we can see the whole model.

So in order to do this we need to go into the /GAME directory in your WCPSO install. We're looking for a file named "Viewobj.IFF", once you find it, open it in WCPPascal. Since we're replacing the Piranha, we're in luck as it's the first entry in the file. You should see a piece of code that looks like this:

Figure 24: The original Piranha entry in viewobj.iff

```
FORM "OBJ "
  {
```

```
    CHUNK "DATA"
    {
     long 16
     cstring "piranha"
    }
   }
```

We're going to want to edit this section of the file. If you look down the file you'll notice several of the entries have a section of code called "**CHLD**" (or CHILD); the CHILD form tells the viewer to render child objects along with the base mesh. The code is pretty straight forward, each DATA chunk nested in the CHILD form has two string entries; the first is the name of the mesh from the /MESH folder that you want to render, the second is the name of the hard point on the base mesh you want to attach the child mesh to. There doesn't seem to be a limit on the number of children you can render.

Figure 25: The CHLD form

```
FORM "CHLD"
 {
     CHUNK "DATA"
     {
      cstring "piranha2ckpt" --Mesh
      cstring "pirengineattach" -- Hard Point
     }
}
```

So we just add a CHLD form to the base Piranha entry and we're done. Just remember that when you insert the CHLD form you need to insert the form between the closing French brace of the initial DATA chunk and the closing French brace of the OBJ form. If you don't have the braces correct, the compiler in WCPPascal will throw an error.

Figure 26: The complete Piranha code entry in Viewobj

```
   FORM "OBJ "
   {
    CHUNK "DATA"
    {
     long 16
```

```
  cstring "piranha"
 }

FORM "CHLD"
 {
  CHUNK "DATA"
  {
   cstring "piranha2ckpt"
   cstring "pirengineattach"
  }
  CHUNK "DATA"
  {
   cstring "piranha2engn"
   cstring "pirengineattach"
  }
  CHUNK "DATA"
  {
   cstring "piranha2lgun"
   cstring "pirengineattach"
  }
  CHUNK "DATA"
  {
   cstring "piranha2rgun"
   cstring "pirengineattach"
  }
  CHUNK "DATA"
  {
   cstring "piranha2nguns"
   cstring "pirengineattach"
```

```
    }
   }
  }
```

That's it! Under Compile in WCPPascal's menu, just select "Compile & Build IFF". You can now load up the game and view your hard work in the ship viewer. Now because nothing is as simple as it seems, we need to modify the Piranha's PIRANHA.IFF in the /SHIPS directory to get it to work in the game. Otherwise you end up with a flying set of wings. So we open PIRANHA.IFF in WCPPascal. Remember to back up the original first!

Figure 27: The original PIRANHA.IFF

```
IFF "C:\Games\Secret Ops\ships\PIRANHA.IFF"
{
 FORM "SHIP"
 {
  CHUNK "TYPE"
  {
   long 3
   long 16
   long 1
  }
  CHUNK "BASE"
  {
   cstring "piranha"
  }
  CHUNK "CKPT"
  {
   long 52
   cstring "piranha"
  }
  FORM "MNVR"
  {
```

```
CHUNK "ATTK"
{
  cstring "attack"
  cstring "chaseattack"
  cstring "attackflee"
  cstring "componentattack"
  cstring "turretattack"
}
CHUNK "MISL"
{
  cstring "launchchaff"
}
CHUNK "GUNS"
{
  cstring "rolloutavoid"
  cstring "pitchavoid"
  cstring "yawavoid"
}
CHUNK "AVDF"
{
  cstring "rolloutavoid"
  cstring "pitchavoid"
  cstring "yawavoid"
  cstring "flee"
}
CHUNK "AVDC"
{
  cstring "capshipflee"
}
```

```
 CHUNK "CHAS"
 {
  cstring "loop"
  cstring "yoyo"
  cstring "barrelroll"
  cstring "flee"
 }
}
CHUNK "PHYS"
{
 float 12000.0          //mass
}
CHUNK "SLID"
{
 long 1
}
CHUNK "HITP"
{
 float 160.0            //hitpoints
}
CHUNK "POWR"
{
 float 100.0
 long 0
}
FORM "PSYS"
{
 FORM "ENGN"
 {
```

```
CHUNK "POWR"
{
 float 100.0
 float 0.0
 float 650.0
 float 650.0        //speed max
 float 500.0        //speed std
 float 150.0        //speed min
}
CHUNK "DATA"
{
 float 650.0        //acceleration
 float 1400.0       //afterb. speed
 float 1800.0       //afterb. acceleration
 float 360.0        //afterb. fuel
 float 90.0         //Y
 float 90.0         //P
 float 135.0        //R
 float 220.0        //YPR rate of change
}
FORM "CONS"
{
 CHUNK "CONE"
 {
  cstring "pir_aft" //mesh
  float 0.0         //X
  float -0.05       //Y
  float -3.9        //Z
 }
```

```
 }
}
FORM "GUNS"
{
 CHUNK "POWR"
 {
  float 100.0
  float 0.0
  float 280.0       //gun energy
  float 45.0        //recharge max
  float 30.0        //recharge std
  float 3.0         //recharge min
 }
 CHUNK "GUN_"
 {
  long 4            //gun id
  long 0            //ammo
  cstring "gun1"    //hardpoint
 }
 CHUNK "GUN_"
 {
  long 4            //gun id
  long 0            //ammo
  cstring "gun2"    //hardpoint
 }
 CHUNK "GUN_"
 {
  long 4            //gun id
  long 0            //ammo
```

```
   cstring "gun3"      //hardpoint
  }
 }
 FORM "MISS"
 {
  CHUNK "POWR"
  {
   float 100.0
   long 0
   float 800.0
   float 580.0
   float 360.0
   float 36.0
  }
  CHUNK "DECY"
  {
   long 24
  }
  CHUNK "LNCH"
  {
   cstring "miss1"     //hardpoint
   cstring "miss2"     //hardpoint
  }
  CHUNK "MIS_"
  {
   long 2
   long 2
  }
  CHUNK "MIS_"
```

```
  {
    long 4
    long 4
  }
 }
 FORM "FSHD"
 {
  CHUNK "POWR"
  {
    float 100.0
    float 0.0
    float 420.0
    float 30.0        //recharge max
    float 20.0        //recharge std
    float 2.0         //recharge min
  }
  CHUNK "DATA"
  {
    float 210.0       //shields front
    float 210.0       //shields rear
    float 0.5
  }
  CHUNK "MESH"
  {
    cstring "c_shld_1"  //mesh
    float 10.0
  }
 }
}
```

```
FORM "ARMR"
{
  CHUNK "DATA"
  {
    float 250.0         //armor front
    float 200.0         //armor rear
    float 220.0         //armor left
    float 220.0         //armor right
  }
}
FORM "EXPL"
{
  CHUNK "TYPE"
  {
    long 21
    long 9
    long 15
  }
}
FORM "SPRK"
{
  CHUNK "DATA"
  {
    long 0
    float 25.0
    long 0
    long 0
    float -5.0
    float 0.1
```

```
  float 2.0
  long 6000
  float 4.0
  float 6.0
  float 0.3
  float 0.5
  float 0.1
  float 0.4
  float 0.01
  float 0.05
}
CHUNK "DATA"
{
  float 26.0
  float 50.0
  long 0
  long 0
  float -5.0
  float 0.1
  float 2.0
  long 6000
  float 2.0
  float 4.0
  float 0.1
  float 0.5
  float 0.1
  float 0.4
  float 0.05
  float 0.1
```

```
    }
    CHUNK "DATA"
    {
     float 51.0
     float 90.0
     long 0
     long 0
     float -5.0
     float 0.1
     float 2.0
     long 6000
     float 1.0
     float 2.0
     float 0.1
     float 0.5
     float 0.1
     float 0.4
     float 0.05
     float 0.2
    }
   }
  }
}
```

So here's where we want to do a little trickery to get the game to do what we want. Many of the models have rotating parts, the SWACS has its rotating raydome, the comms station has a rotating ring. We just want to use that form to assemble our own models. The form we want to use is the PART form. The form is actually very similar to the CHLD form above, with a few additional options. The first difference is the order of the entries, in the PART form, the hard point is the first entry and the mesh name is second. The second difference is a long entry, this indicates the rotation of the object in

degrees per second. Now we don't want a revolving cockpit or guns, so we want to set the long to 0. This tells Vision that the object should rotate at 0 degrees per second.

The PART form appears near the end of the ship file, it is inserted between the ARMOR form and the EXPL form. So we do the same exercise as before, insert a PART form and add as many DATA chunks as we need to assemble the model.

Figure 28: The SWACS PART form

```
FORM "PART"
{
 CHUNK "DATA"
 {
  cstring "domeattach" -- Hard Point
  cstring "awac_top" -- Mesh
  float 60.0 -- rotation in degrees per second
  long 0
  long 0
 }
}
```

You might also need to modify the CONS form, which tells Vision where to position the thruster exhaust mesh.

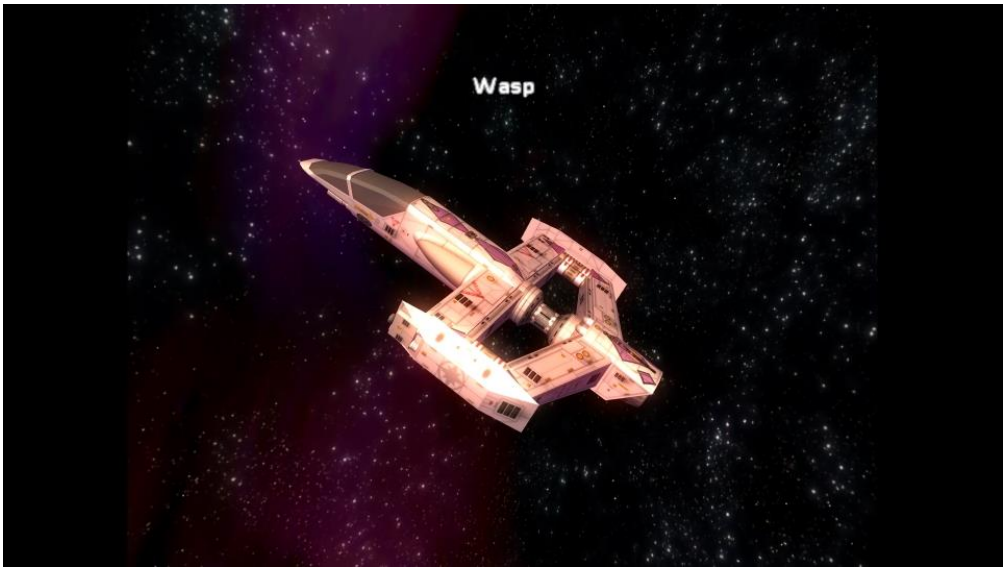
Figure 29: the CONS form

```
FORM "CONS"
{
 CHUNK "CONE"
 {
  cstring "pir_aft" //mesh
  float 0.0       //X
  float -0.05      //Y
  float -3.9       //Z
 }
```

Remember when we placed that hard point called “Engine”? We did that so we could remember where to position the mesh in the CONS form. Clever, eh? You can either check the coordinates in your 3D software, or open the mesh in WCPEdit. Once you have them you just change the X,Y,Z entries in CONS to match.

Once you’ve done that compile and build your new ship.iff and you’re ready to fly!

Figure 30: our new F-106-C ready for action



Hopefully this tutorial was useful and will help you start making new custom content for WCPSO. If you have questions or feedback, feel free to PM me or start a new thread in the forums at the Wing Commander CIC. Note that this is a tutorial for a vanilla install, with patches like WCP Unlimited, the largest constraint, that of 1,500 triangles per mesh was removed. If you’re developing content for an unlimited install, you can omit sectioning the model and modifying the viewobj and ship .IFFs, however meshes larger than 1,500 triangles break WCPEdit so you’ll need to do more manual work in positioning hard points and modifying weapons load outs. I will post additional tutorials with how to build fighters with EPODS and turrets, as well as some capital ship tutorials in the near future. Enjoy and happy modding!