Introduction:

This guide is a full guide to any and all 3D gfx in Paradox Interactive games. Most of the examples will be from my HOI4 mod: Old World Blues. Future updates may include 2D gfx, but this is meant as a general guide to the Clausewitz Engine, the 3D backbone of EU4, HOI4, and Stellaris. The guide will be following the protectron asset in OWB through its process. This entire guide will be using Blender 3D, Notepad ++, and GIMP. Any questions can be sent to me on discord via https://discord.gg/2W8JpRP

Any and all errors or outdated information can be pm'd to me so I can fix it. Thanks!

A link to the blender plugin github https://github.com/ross-g/io_pdx_mesh go to the "Releases" section to download a zip, and simply in blender use install addon -> select the zip.

Part 1: Modeling

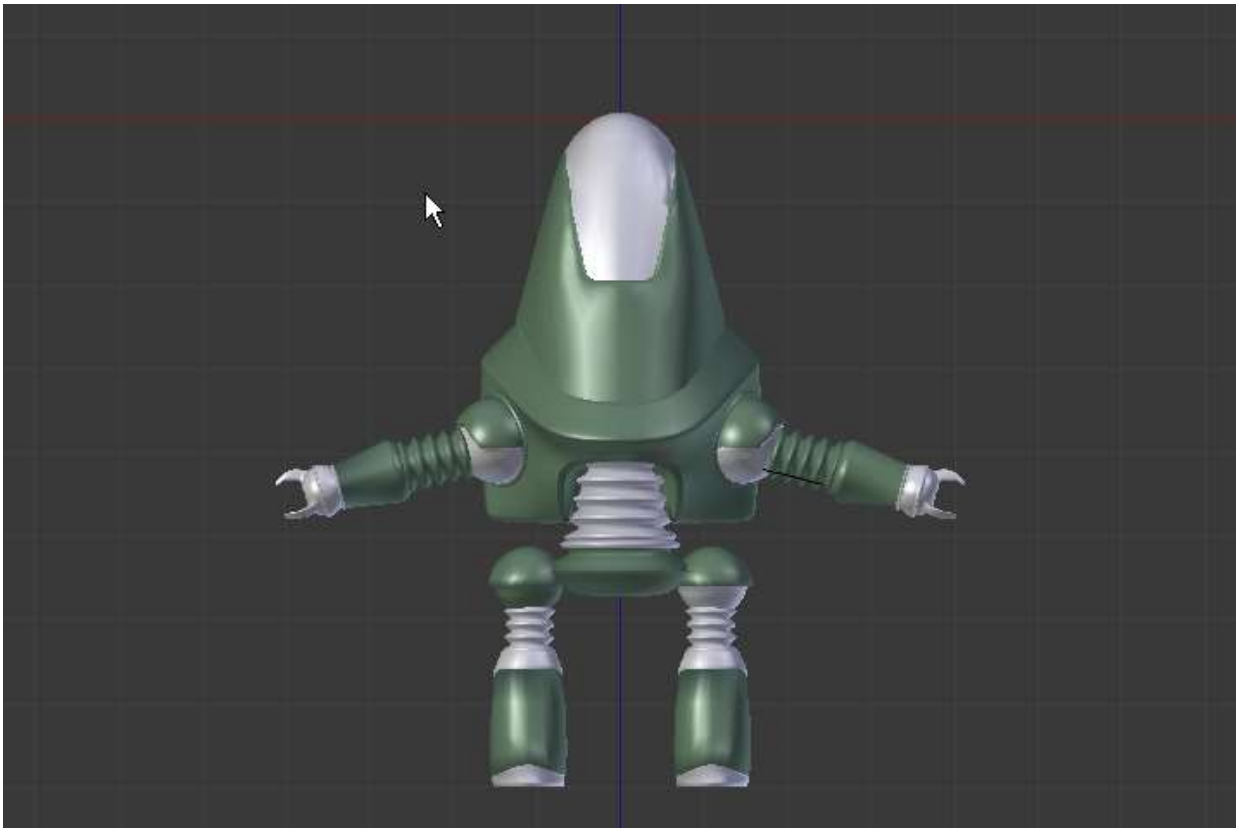
The first section of this guide will in-depth cover modeling an asset and various approaches to preparing something that is detailed to being added into the game while being an efficient use of user resources.

Highpoly / Detailed Mesh

The first step to adding something to a game, any game, is to get a highpoly or detailed version of the mesh completed. This section is going to be a majority theory.

The most important thing to understand about this process are that there are many approaches which each give varying levels of product. Each also takes different skills. My preference leans heavily towards vertex manipulation, which is using the various vertices that are the foundation of 3D models, and manipulating them as opposed to adding a filler step, such as digital sculpting. I find vertex manipulation easier than sculpting for my projects, as I make many more hard surface objects, and do not have a great drawing tablet at my disposal.

Here is the protectron after I sat down for a few hours with a reference image, showing both front, and side profiles in orthographic projection (without perspective).



To give a quick rundown on the production process, it is modeled with a mirror, subdivision and subsurface modifiers, the tubes are array modifiers bent to a curve.

This is the most subjective step in the process.

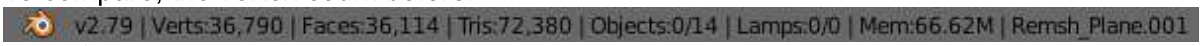
Lowpoly / Game Ready

The lowpoly mesh is up to some debate between me and another promident 3D artist in the PDX community, the author of Star Trek: New Horizons. I use lowpoly models in game, because I believe the efficient use of hardware is important, and it is more helpful for animation. Using highpoly and detailed meshes in game uses more resources, but produces much higher quality visuals, as would be expected. Striking a balance where it is hard to tell a lowpoly mesh isn't highpoly is the best, but more time consuming.

To create a lowpoly mesh, the highpoly mesh needs to undergo retopology, or making a "skin" over it with uniform, optimized geometry. There are various tools to do this, but I always do it by hand, starting at the largest chunk, and working out. It takes a long time, but I always end with exactly what I want, and can fully optimized my vertex count to whatever I wish.

This process is also subjective, so here is an end result:

To compare, the vertex count before:



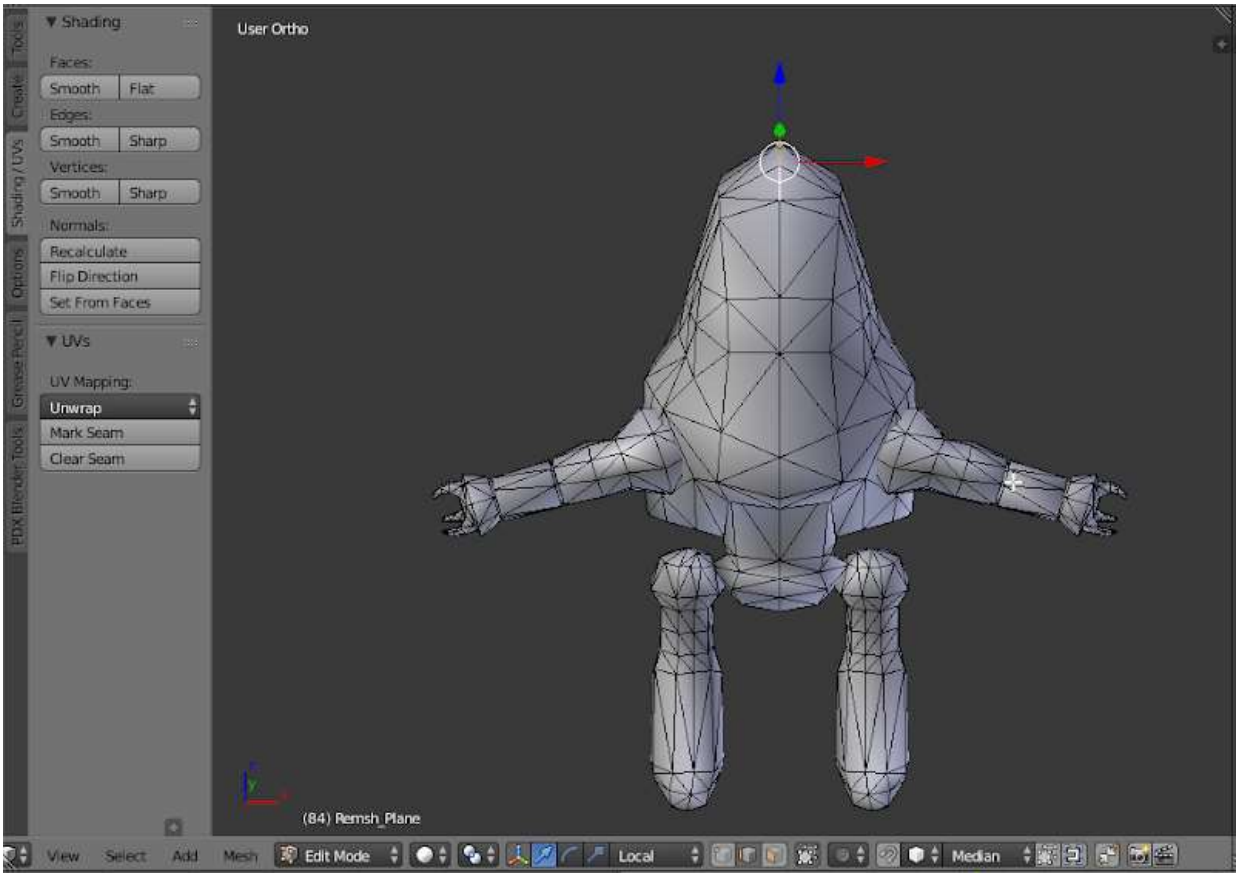
And after:

A huge save on performance, though with one on screen, the impact will be very little, however with 1000 on screen, this is the difference between 60 fps and a crash.

UV Unwrap: The first step to textures

Applying textures is the most necessary step to creating a beautiful 3D asset in any game, as will be apparent near the end of this part. A UV is a set of 2D coordinates that map your 3D vertices onto the 2D plane. Instead of X,Y,Z, we use U and V. Thinking back to elementary school, this is what a complex "net" is, except instead of just making generalized projections that are absolute in scale (1 pixel on a 50 pixel square is 1 unit on an equivalent 50 unit tall face). We create projections of complex shapes that are not (texture stretching). Stretching is 100% avoidable, but by avoiding it, we create seams where two adjacent pieces in 3D are not adjacent in 2D, which can make texturing hard. Striking a balance is important. Adding seams where they are less visible, and adding enough seams to minimize stretching. A highly skilled 3D artist can have almost 0 stretching and hidden seams, but a highly skilled texture artist can work around both stretching and seams, so a balance should be struck if one section has to make up for another. Also it is worth noting that lower polygon meshes are much harder to eliminate both issues on.

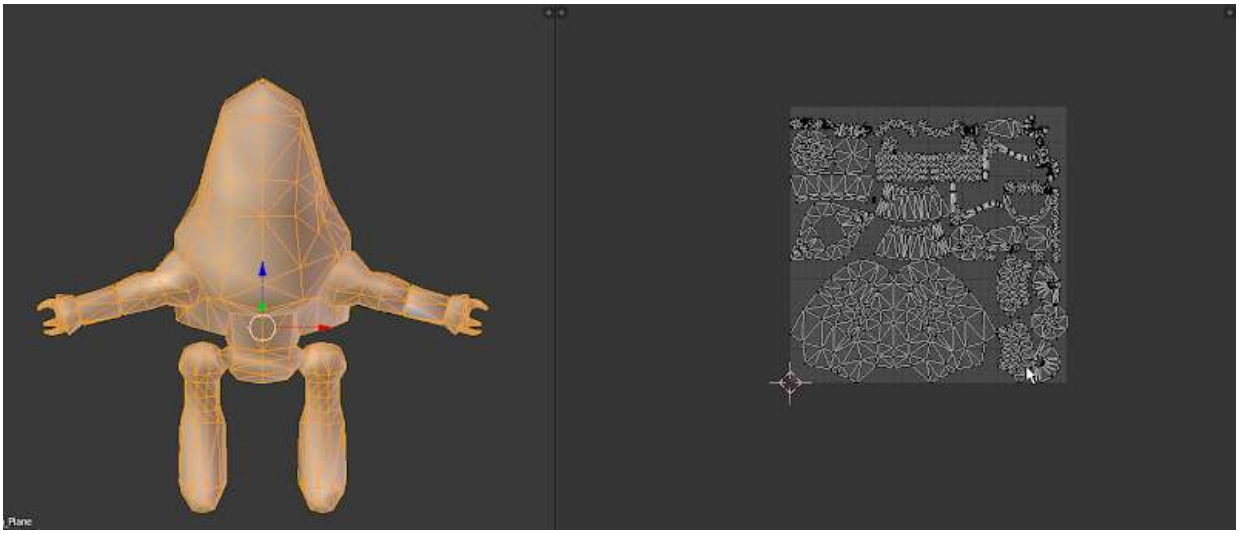
To UV Unwrap, go into edit mode on the lowpoly model. Click edge select mode at the bottom, and select your edges you want to be seams.



Then press mark seams.

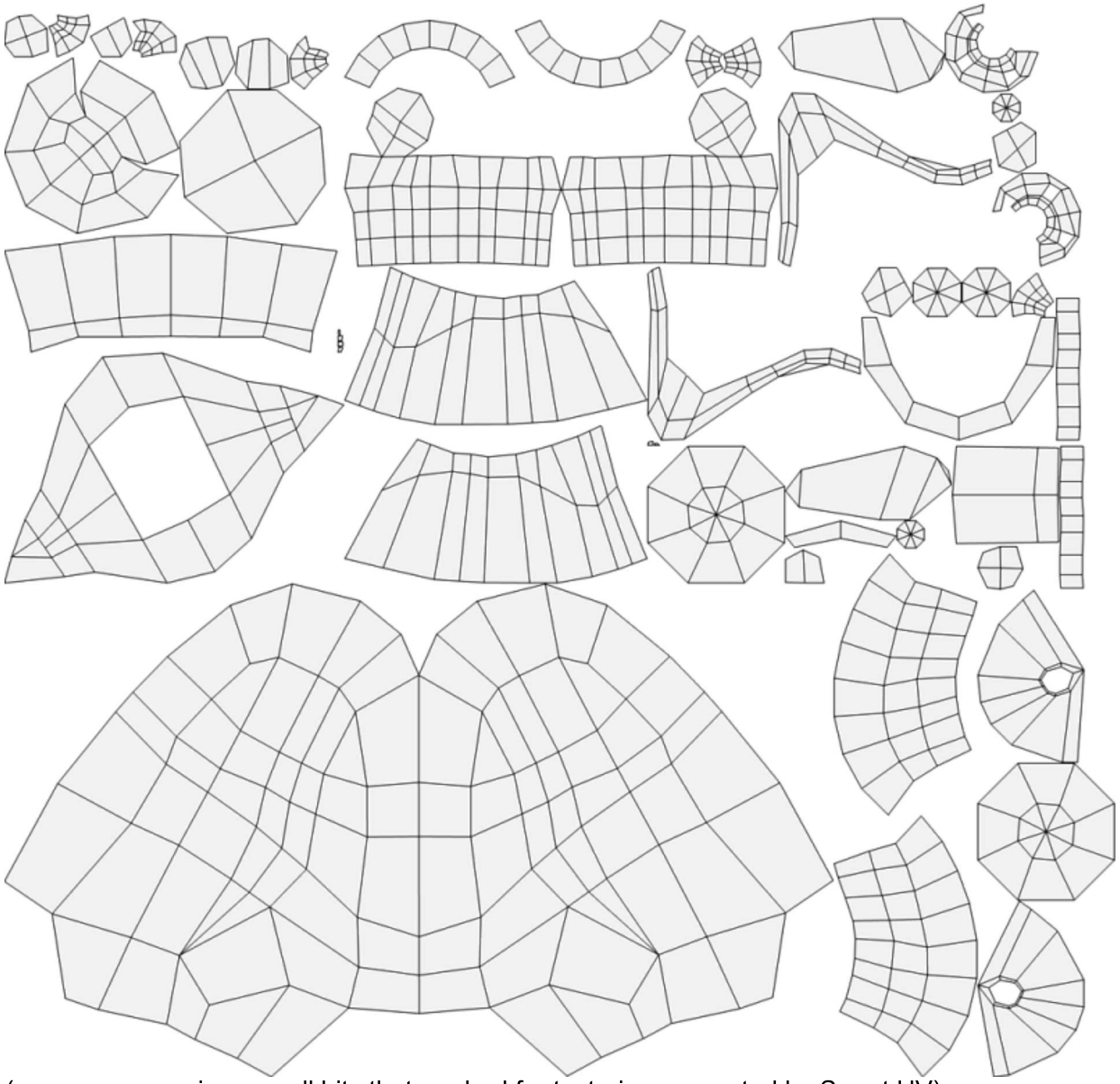
After marking all your seams, press A until the model is selected, then U and choose unwrap. You can also ignore marking every edge seam, and use Smart UV Project. It does a decent job, and it was I used here, because stretching was not a huge concern, being that the model is mostly sectioned into materials already.

Here is my UV Map:



Save this as a UV map to use to create textures later is advisable.

To get this image:

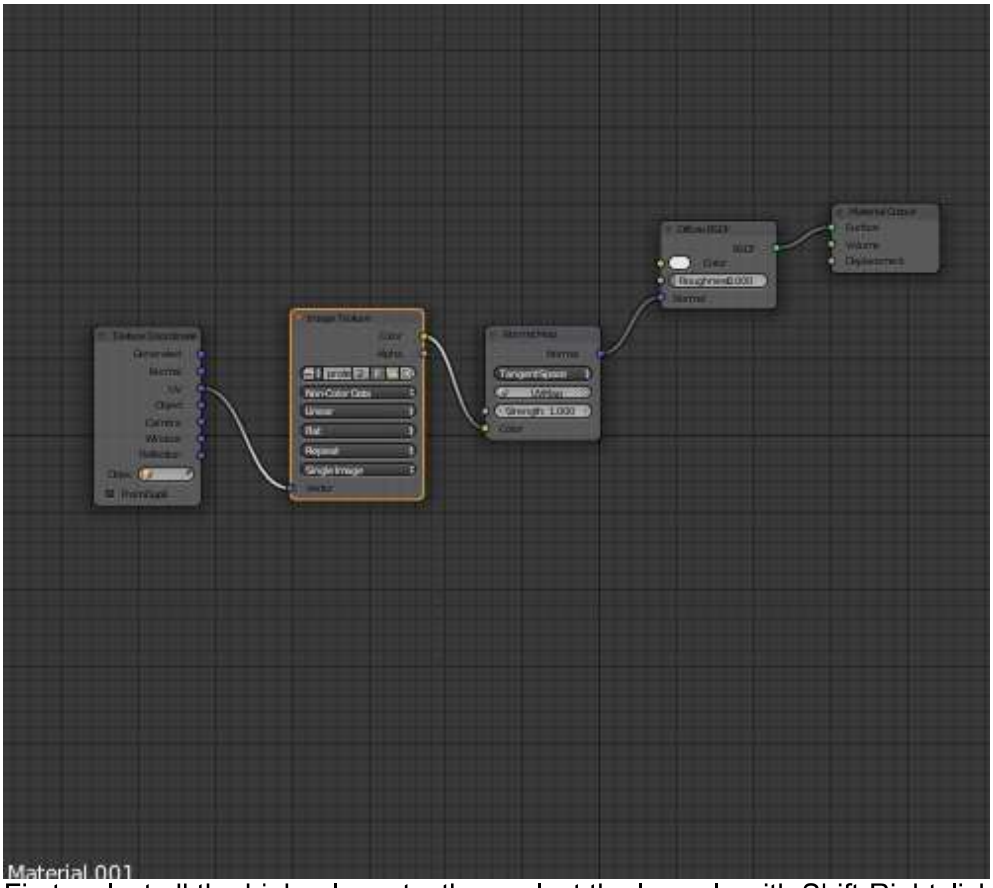


(you can see various small bits that are bad for texturing generated by Smart UV)

Normal Mapping: Making Low Like High

To make sure the lowpoly mesh is as close to highpoly as possible, the next stage to getting it in game is to normal map it. This makes a 2D vector map using the three color channels RGB to make a normalized set of instructions for the graphics card to “push” the appearance. I am not an expert on the foundations of how it functions, but it makes beautiful results. This is a “base” normal map, which I would then add the texture of whatever material I would paint later in the texture.

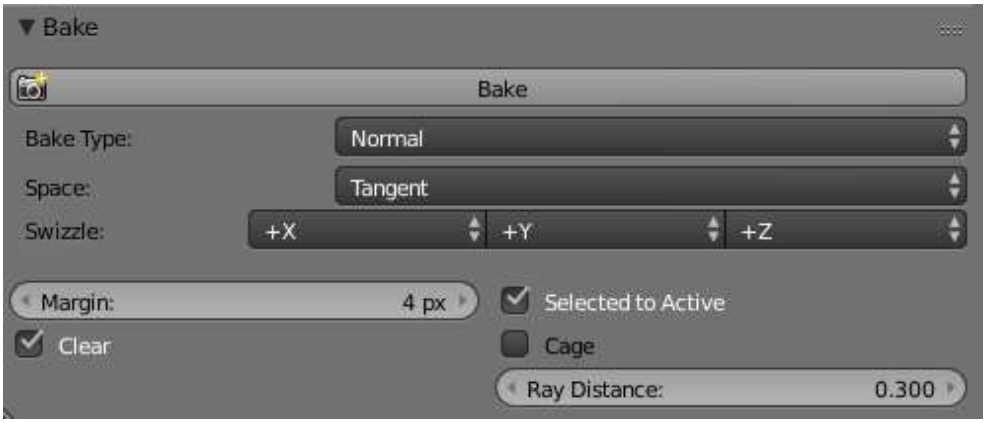
Swap over to Cycles Render. Make a material on the lowpoly with an image texture node, and add a blank texture of the resolution you want. Select the node.



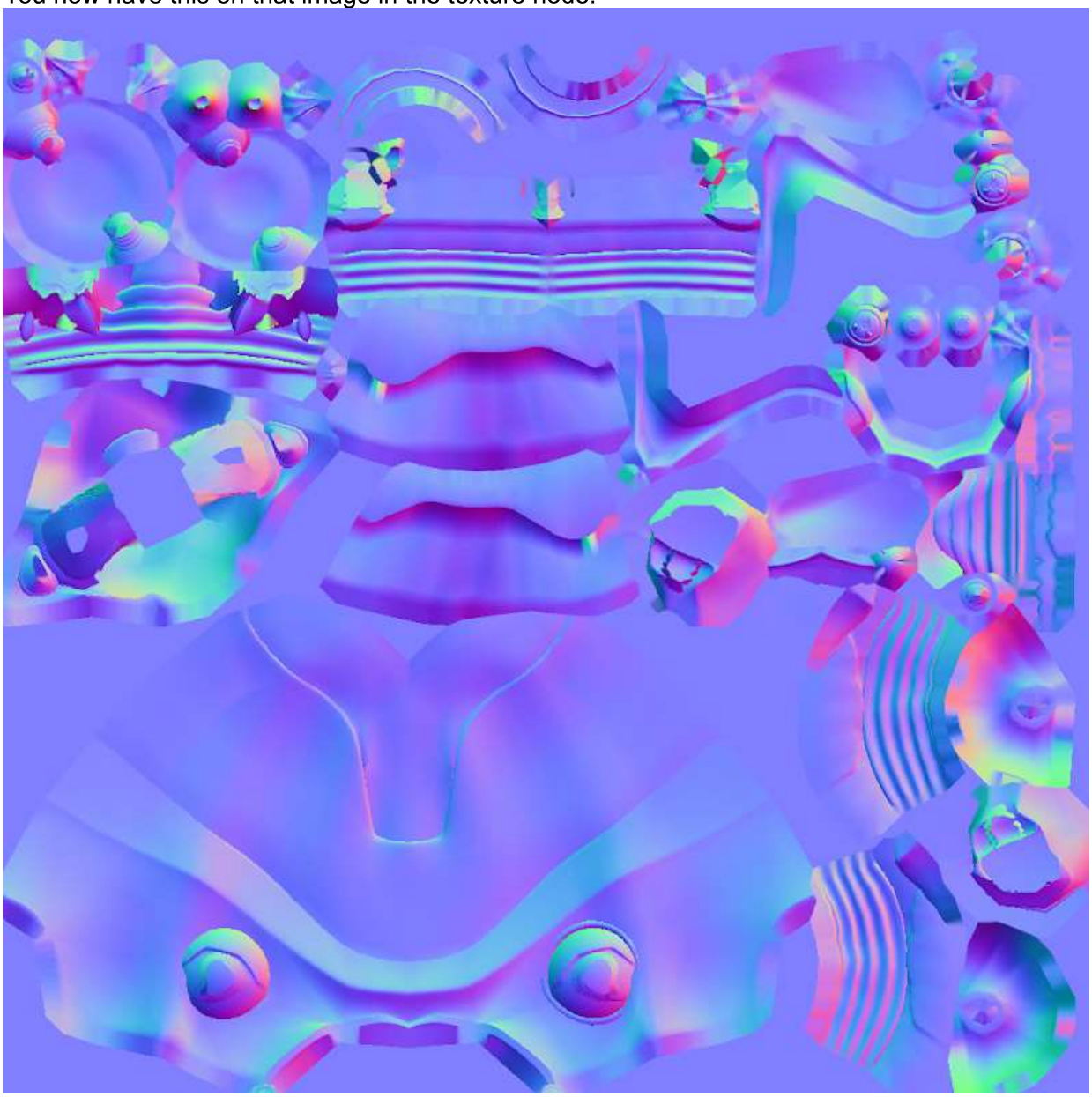
First, select all the highpoly parts, then select the lowpoly with Shift-Rightclick. It is important that it is only one object.

In the Camera properties panel (right), scroll down to Bake.

Copy these settings.
Explanation:
Space: Tangent - this makes a tangent space map, which uses the tangents to the model to define the color of each pixel with their vector.
Margin: 4px - margin around the UV to also bake the color of the edge, helpful to make sure of no gaps.
Selected to Active: VERY IMPORTANT - makes the selected parts bake onto the LAST selection
Clear: Just good practice, if baking multiple sequential parts (different lowpoly meshes) uncheck this.
Swizzle: leave this alone, it defines how the xyz are mapped to rgb.



Bake!
You now have this on that image in the texture node:



Texturing

This step is the most important for making a fully fledged asset. It breathes life into the blank white shape you have on your screen right now.

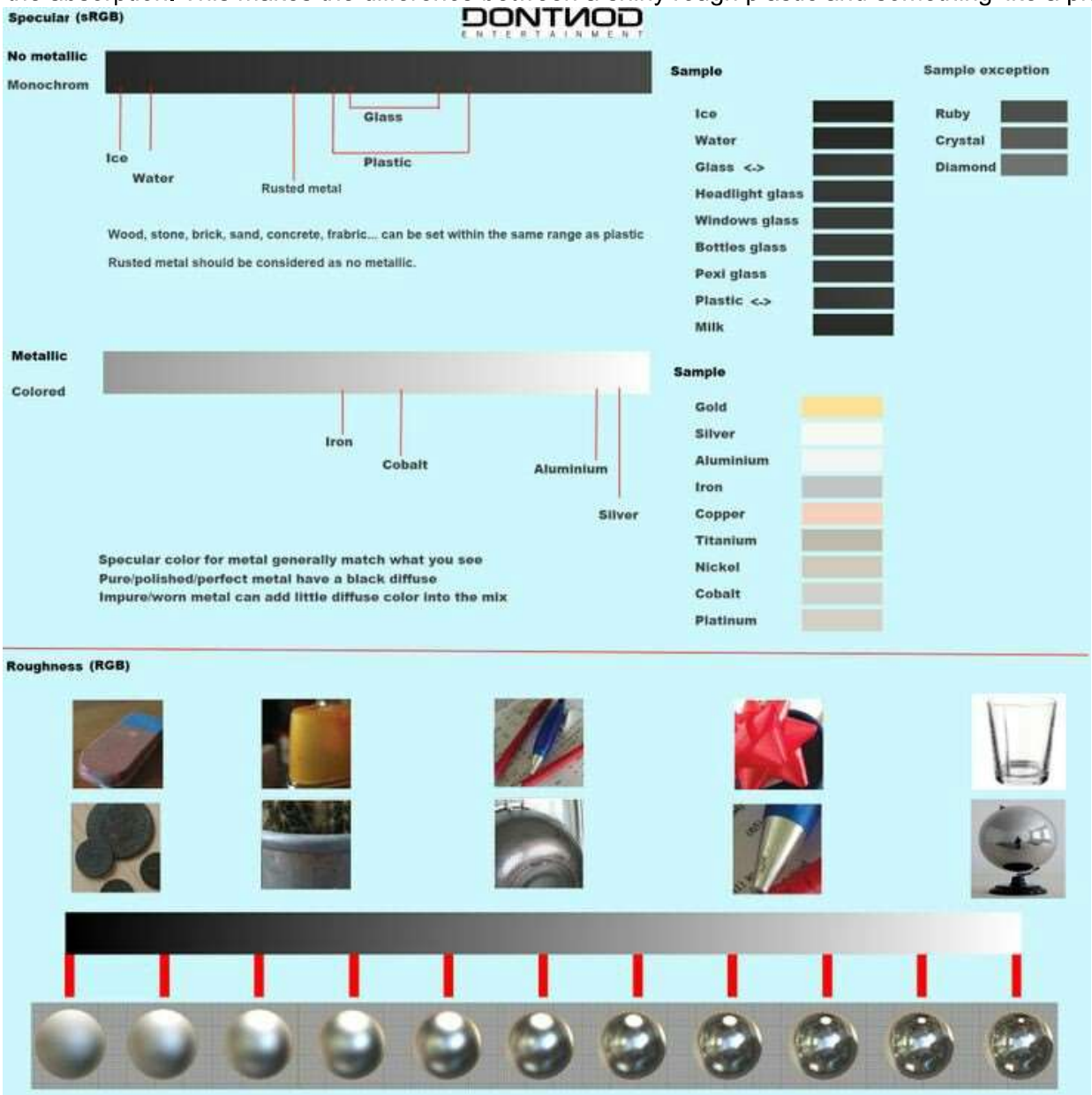
Disclaimer: I am not the texture artist for OWB and send everything our texturer needs to create the textures, but I do understand the process.

To make a texture, import the UV layout as pictured above into a photo editor. Make a new layer below it, and make the UV layout transparent. Import the normal under the UV layout.

You can begin texturing. This is also very subjective, so I can only explain how Clausewitz interprets the textures. You only have to remap each channel to the below, or use the photoshop exporter (not detailed here).

A guide to a Clausewitz texture set:

Input textures:
Diffuse/Albedo - raw color data, only shading material specific shadows.
Normal - explained above, gives more detail for graphics card
Specular - details interaction with light
Specular - how reflective an object is
Metalness - If the object is metallic (metals have special interactions with light when compared to other materials.)
Roughness - how rough an object is, this affects the direction of the scattered light while specularity affects the absorption. This makes the difference between a shiny rough plastic and something like a piece of paper.



Here is a guide to specular maps:

Output maps (Via Stellaris Wiki):
Diffuse:
R - Diffuse R
G - Diffuse G
B - Diffuse B

A - Opacity (Alpha)

Normal:
R - Normal R
G - Normal R
B - Emissive
A - Normal G
(Actual normal is 2D, not 3D as we exported so the blue channel is lost)

Specular:
R - Mask (custom empire colors in stellaris, etc.)
G - Specular
B - Metalness
A - Gloss (Roughness inverse aka 1 roughness is 0 Gloss)

The output textures for the protectron:
Diffuse:
Normal:
Specular:

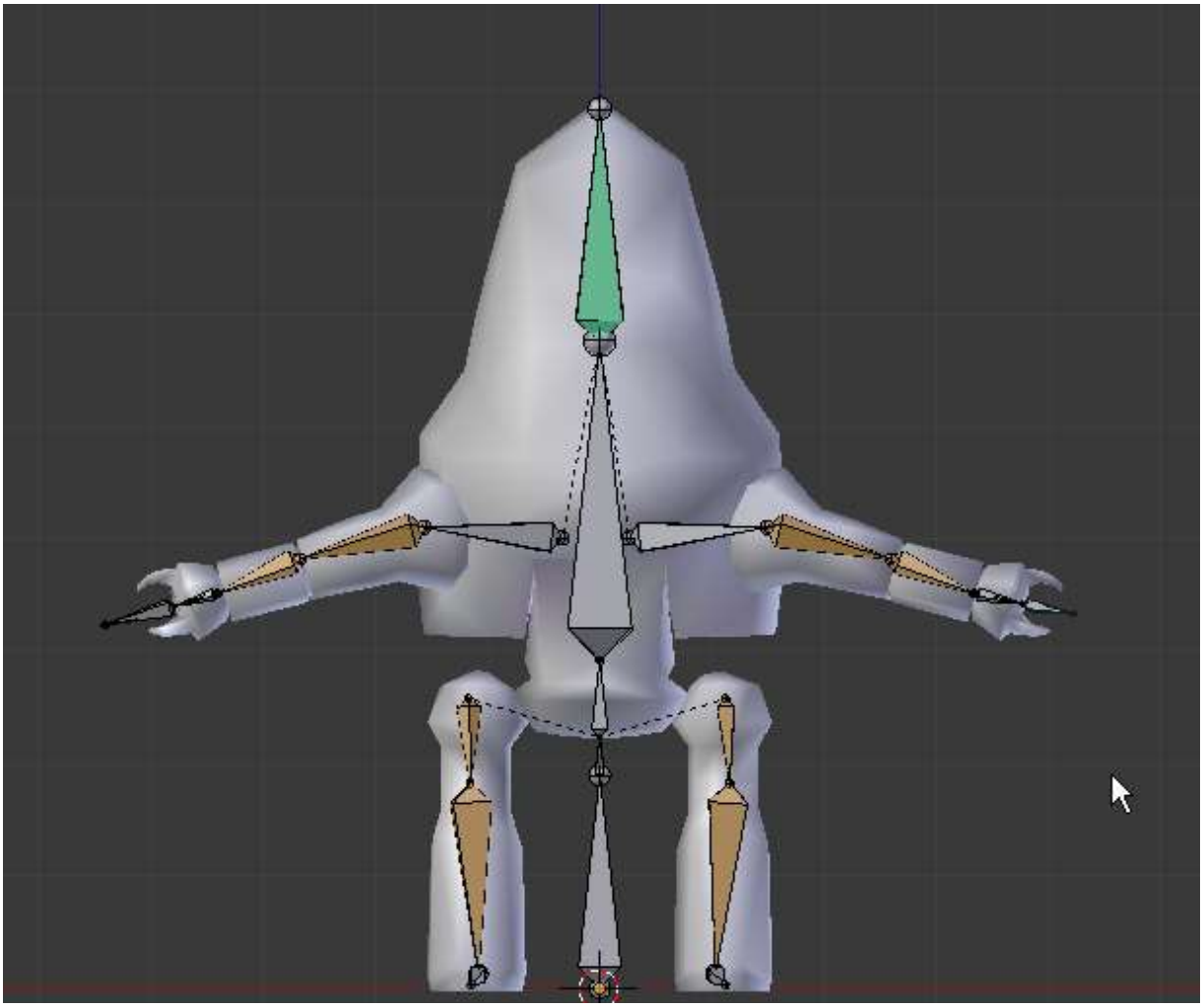
Part 2: Rigging

This is where we make that model dance! Rigging is an extremely complicated topic, so this will be very abbreviated compared to what one can do. The only important thing to remember is no shape keys, no drivers, and no zero-length bones. Shape keys take two mesh states and interpolate between them. For example, neutral state face with a shape key for a smile. A driver changes some set value in blender with a bone position. Clausewitz supports neither.

FK / Deform Rig:

The FK rig is what actually moves the mesh. The mesh is directly bound to this rig, and the rig is what we end up exporting in the end. Two rules for this: Make sure every bone has a parent except one named Root. No bones where the head and tail are the same (zero length bones).

The basics to rigging are to create an armature, and add a bone at the bottom between the feet. Press E to extrude a bone from the tip of the previous one. That creates a join between your new bone and old bone where it can bend. Make a joint at each natural joint of your model, until you get something like this:



Important to make sure the bones are fully aligned, front and side, in the middle of the model, with the joints where the natural joints are. Ignore the colors, we get to those later.

Make sure to add “nodes” at the end of hands or on the back for some models, they are used as attachment points for things! A note on stellaris: Stellaris uses locators to position guns and ship sections. These are currently unable to be added to my knowledge with the technique I use to export, through an upcoming plugin may change this (workaround in the exporting section).

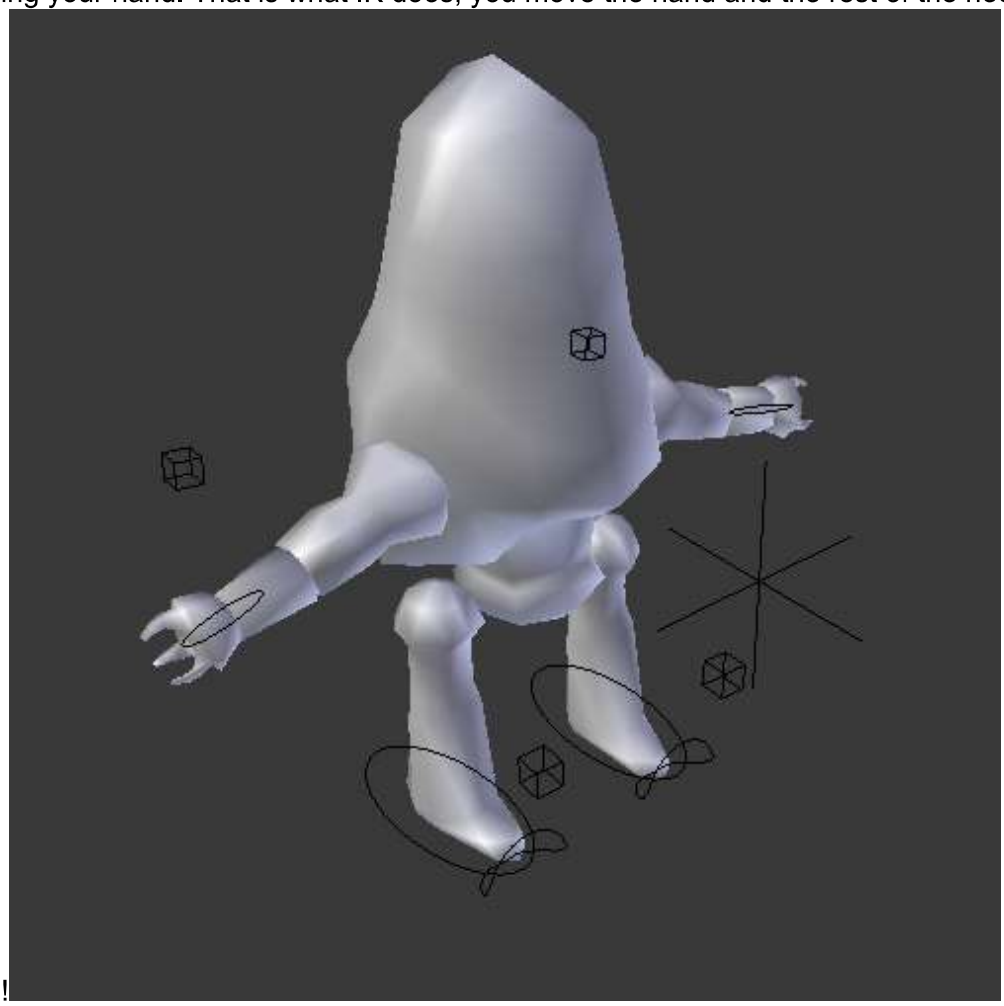
I organized my bones into bone groups, to keep track of them, and better be able to animate from orthographic views (if they are the same color it’s hard to tell right from left).

Next we need to weight this mesh to our new set of bones! Select the mesh then the armature in object mode. Press Ctrl-P and select With Automatic Weights. This will make vertex groups that assign each vertex to a group with the same name as a bone. Each vertex is weighted between 0 and 1 for influence with that bone. This can be changed by selecting the mesh and going into weight paint mode. This process can be finicky, and much better tutorials exist covering this than I can.

IK: Inverse Kinematics

The next step is to create some controller bones to control our future IK or inverse kinematic rig. Inverse kinematic uses a controller bone at something like the hand, to move the whole arm. Think of it like this: you don’t move your shoulder, then your upper arm, then your elbow, then your lower arm then wrist then hand to grab a glass, you move your hand to the glass! If you think, you do have to move all of those, but only think

about moving your hand. That is what IK does, you move the hand and the rest of the necessary stuff follows right along!



So for a bipedal character like this, I make a few basic controls:
Hands
Elbows
Look target (allows you to turn the head to a “target” empty, gives the impression of focus
Feet
Knees

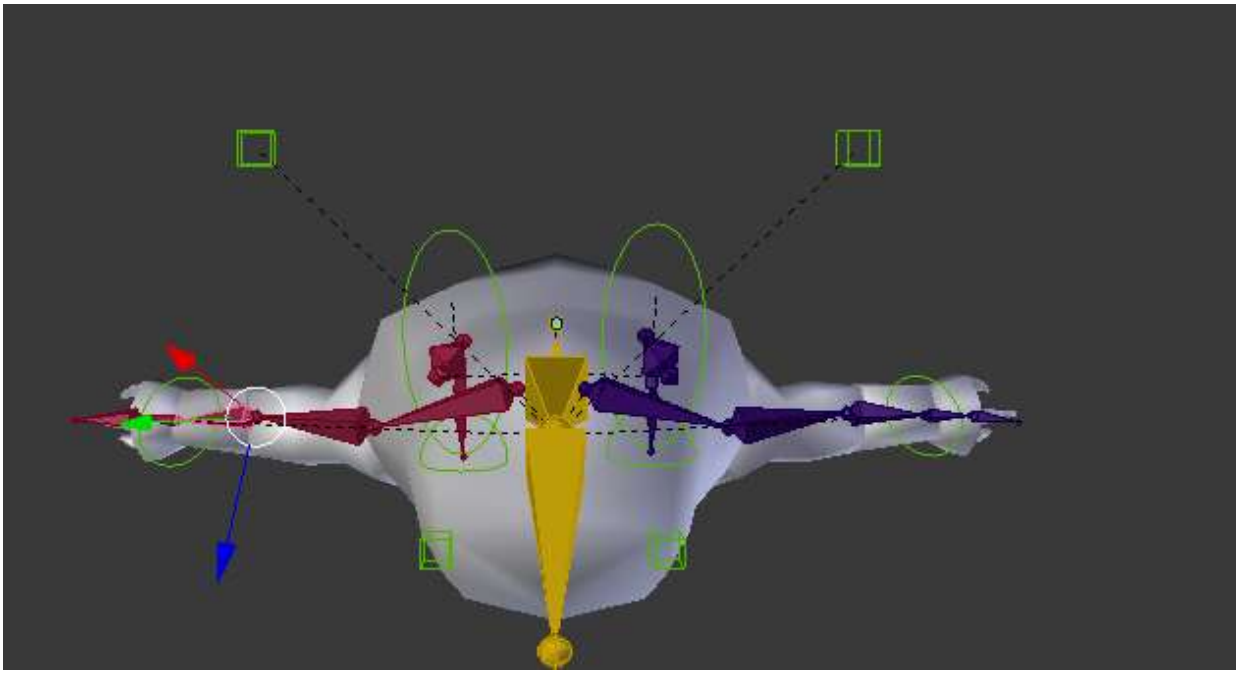
One control that is popular to avoid is a heel-roll rig, they contain zero length bones, but not in blender’s system where bones can be parented but separated. In Clausewitz bones are end to end, which makes heel-roll zero length bones!

Here are my IK controllers

To make them function:
Hands:
Select the forearm bone, add a bone constraint Inverse Kinematics. Select hand IK controller from dropdowns (select armature > opens bone select menu)
Set the chain length to just below the shoulder
Check Stretch
Make sure the head of the IK control is touching the tail of the forearm
Select the hand bone, add copy rotation, select the IK hand controller.

Use same steps for feet

Elbows:
Select the upper arm bone, add Inverse Kinematics. Select the elbow IK
Chain length to just below the shoulder (normally 1)
Make sure the elbow is positioned behind the elbow joint in edit mode:



Repeat for knees but in front of the joint

Look:
Select the head. Add track to. Select look IK target
To: Z only

You now have a functional IK rig! Try it out!

Part 3: Animation

Animation is subjective, so I will give instructions in the most general terms.

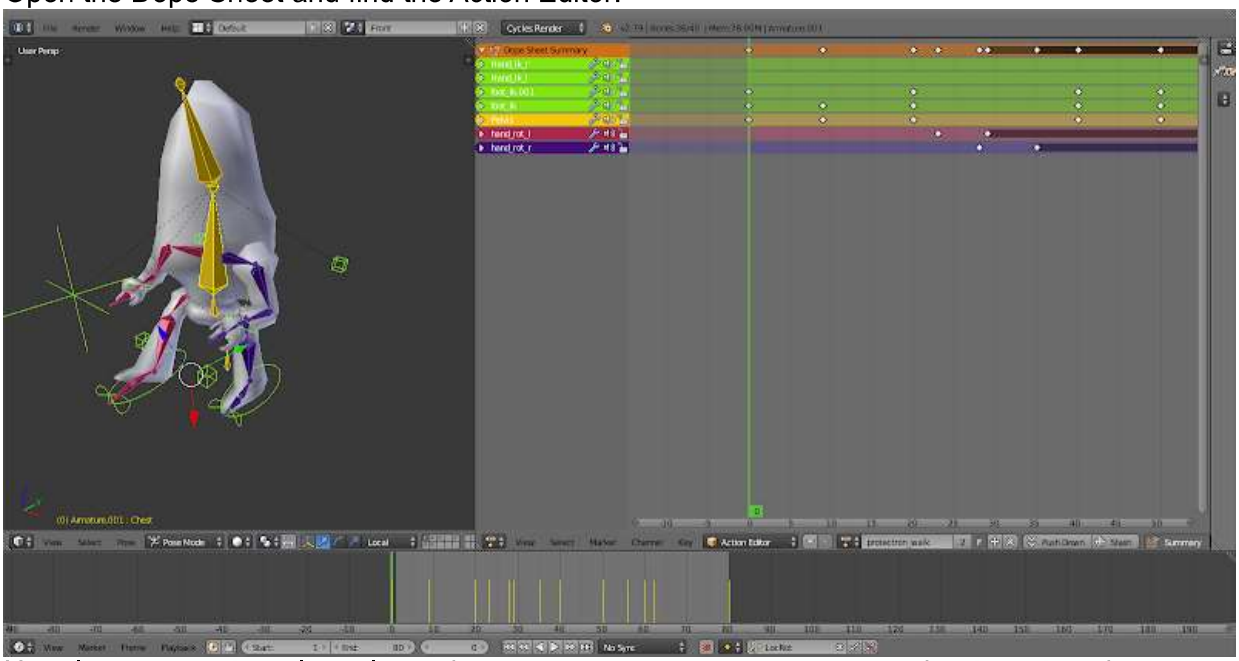
First, find a start position for the rig. The first frame of whatever position you'd like. Find the dropdown under the timeline and select "LocRot." On the right hand side of the 3D Viewport, change your rotation settings to Quaternion WXYZ, this will help make sure your export is exactly what you see, as PDX anim format uses WXYZ.

To my knowledge scale works, but changing the scale of something is rarely recommended unless you intend to create animations with a lot of flow that don't loop seamlessly. (For an example of this I recommend League of Legends: A Twist of Fate). These animations we will make stray away from the fundamental principles of animation, because they aren't for story telling, they are made to loop constantly and not create a ton of visual flash and clutter.

To keyframe, select a bone and press I. Boom! You're on your way.

After making an animation of desired frames, constrain it to those frames. If your animation ends on frame 30, make sure start is 1 and end is 30, not larger or smaller.

Open the Dope Sheet and find the Action Editor:

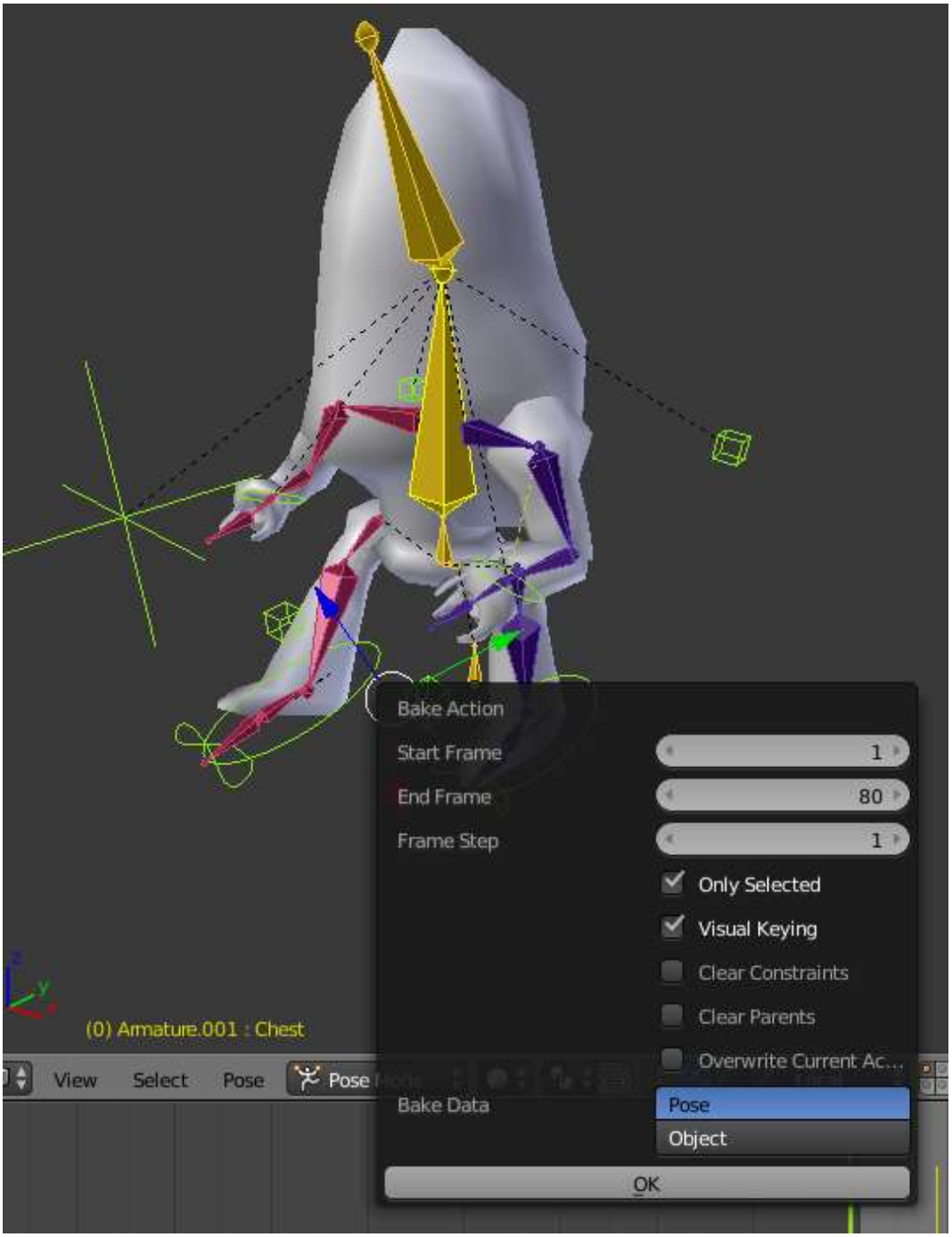


Here I have a basic walk cycle made, you want to name your action on the right side above the timeline and make sure to press the capital F to save it.

Preparing to Export Animations

The first step is to make your animations formatted to be exported into the Clausewitz engine without causing it a meltdown. First we want to bake the actions, This records the position of every bone every frame it moves instead of just the keyframes.

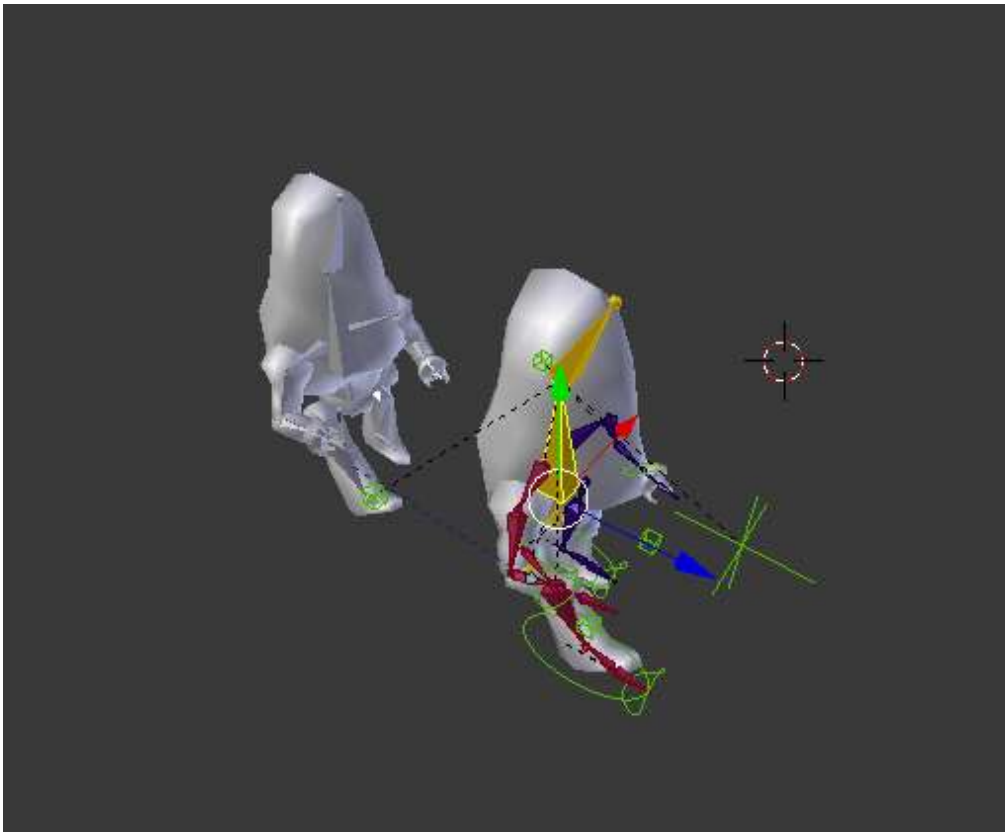
Put your cursor in the 3D viewport, and press space. Navigate to Bake Action and copy my settings.



Make sure to also click the little F and name your new action!

In the Action Editor with your new action, press Shift-O to sample the keyframes, and put one every frame, even if the bone doesn't move.

Next we need to make an export rig. This will make our rig function and not crash horribly. In object mode, copy the rig you have with Shift-D. Right click and go into edit mode. Delete all the IK bones. Bake your action again, but this time check clear constraints and overwrite current action, this removes all traces of IK. It may seem counterintuitive, but once all of your animations are baked once on the IK rig, you can go to the action editor and select a baked animation while in pose mode with your new export rig, and it works great!



Here is my Ik rig in front, and my bake rig behind, both using a baked animation.

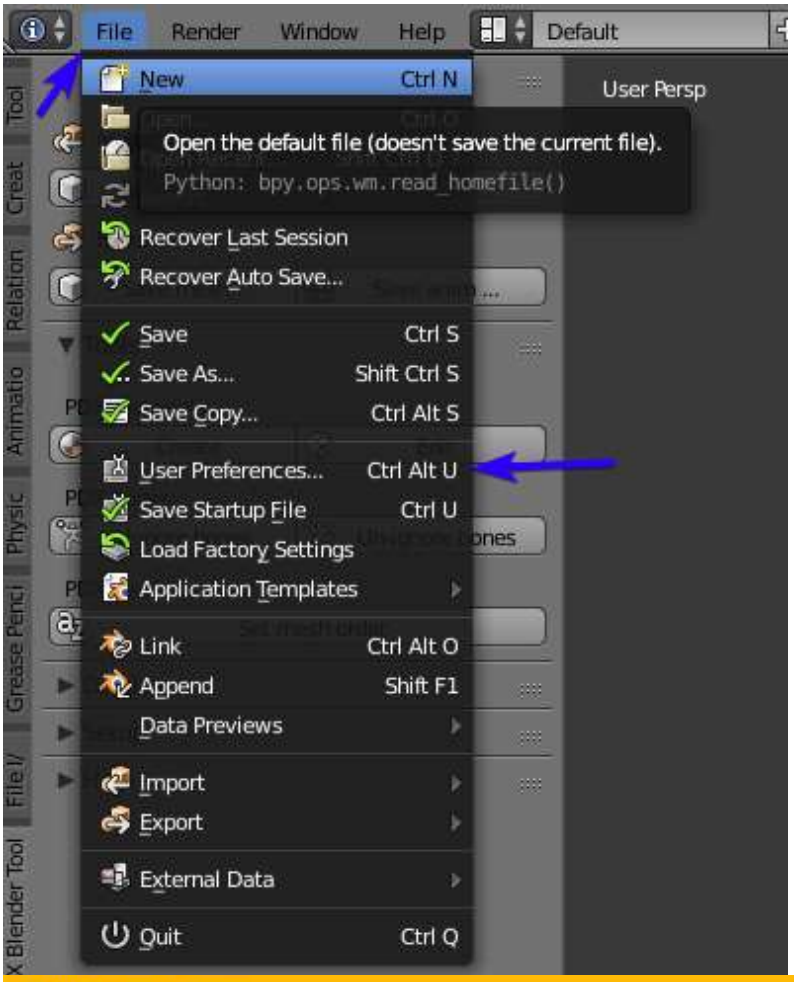
All set to start putting our beautiful model in game!

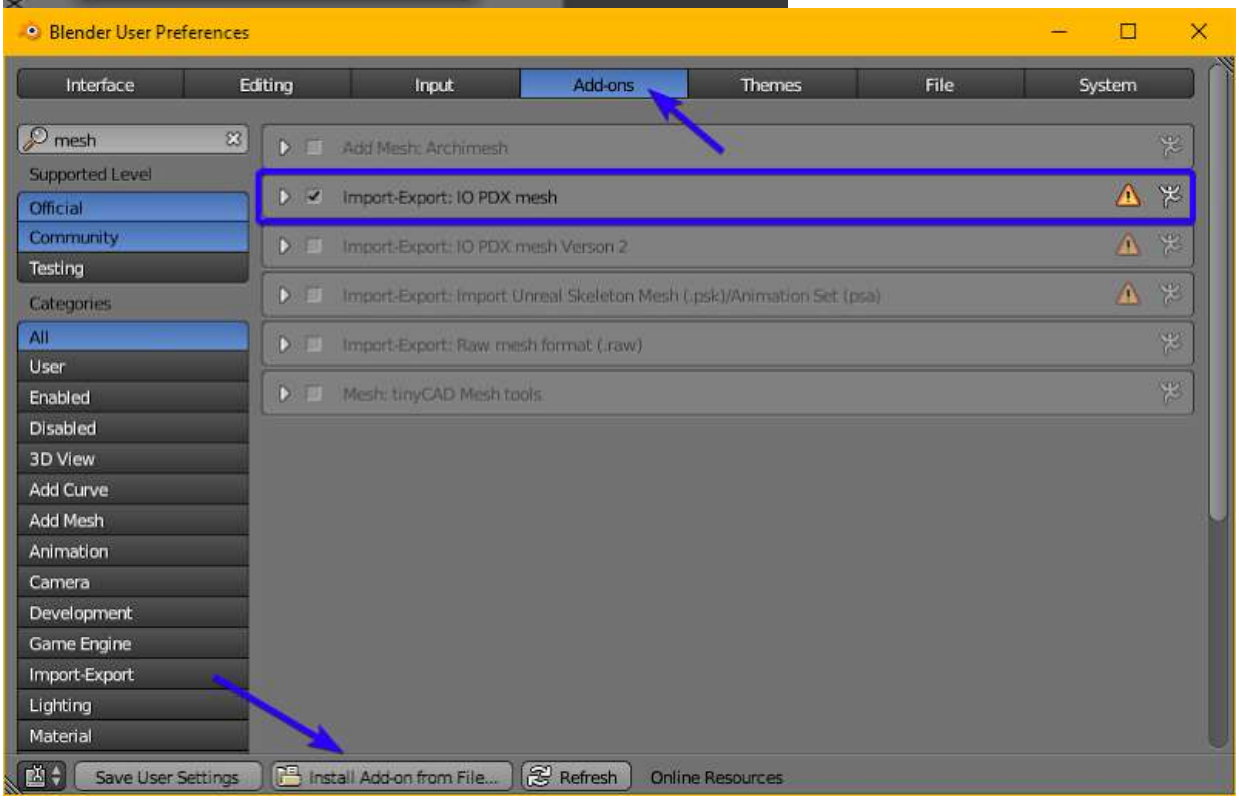
Part 4: Export

As of May 2019, we can now use a plugin inside blender to export directly to PDX mesh and PDX anim files! This also has the benefit of allowing in-file application of a shader object, mesh indexing, and more complex rig setups.

Plugin Installation

You can find the plugin here: https://github.com/ross-g/io_pdx_mesh
Below are the install instructions





PDX Material Creator

The first step to export is to create your PDX Material for export. To do this we go to PDX Material > Create. Give it a name you will recognize, then assign a shader to it. These shaders can be found in your “(game)/gfx/fx/pdxmesh.shader” file at the bottom. Normally you need PdxMeshAdvanced, though sometimes other shaders or custom shaders are needed. Make sure to assign the new material to your mesh.

This material can be edited by selecting the name, and clicking edit in the panel on the left as well.

Export Mesh

To export a mesh make sure you have your mesh selected. Make sure it has your material applied. I prefer to have one export PER FILE otherwise you can get some accidental exports.

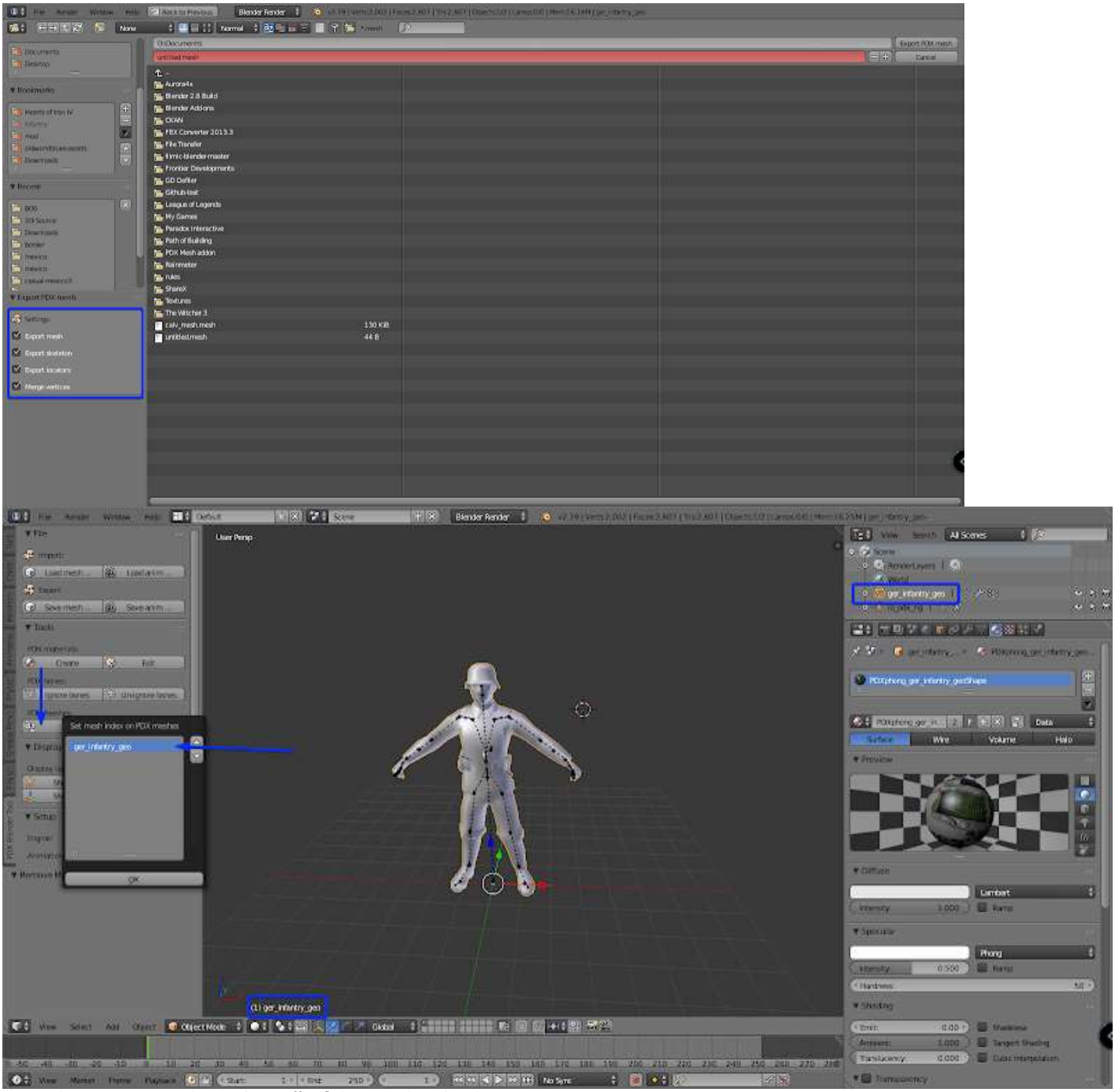
Now is the time to index your meshes! Use this handy button here to index your meshes (starting at zero) so you can reference them later in the .gfx file.

You can see the names of objects in the outliner or in the bottom left by selecting them.

Make sure your orientations are as follows for optimal results.
Apply all transformations and rotations (Ctrl-A) so your object's local position is world origin, with no rotations.
Face everything so that Y- is FORWARD and Z+ is up.

For planar meshes (portraits) you need to make sure your normal is facing Y-, with transforms applied!

Now we just click export mesh, with the mesh selected (not the armature).

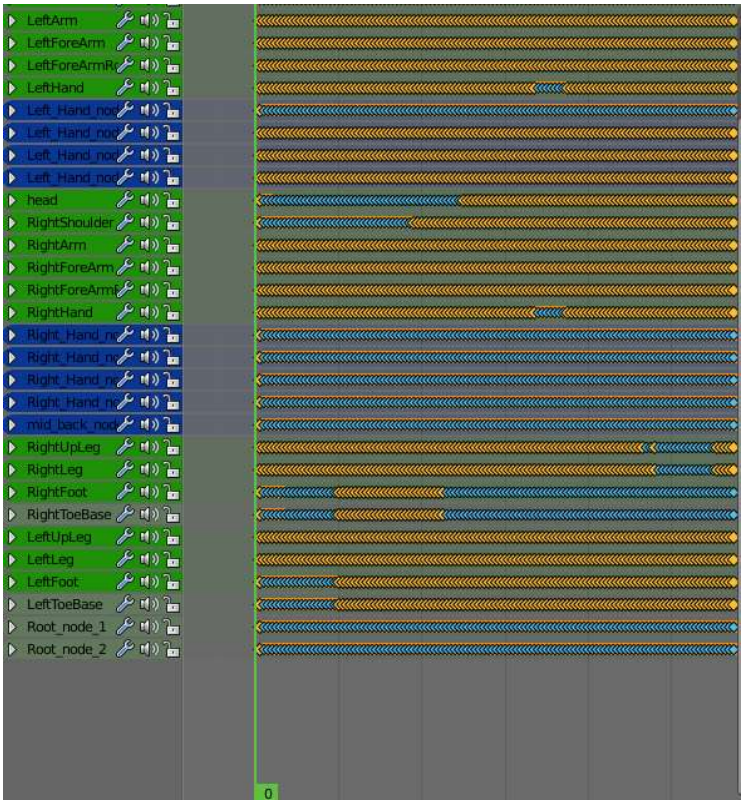


I much recommend leaving all of these checked.

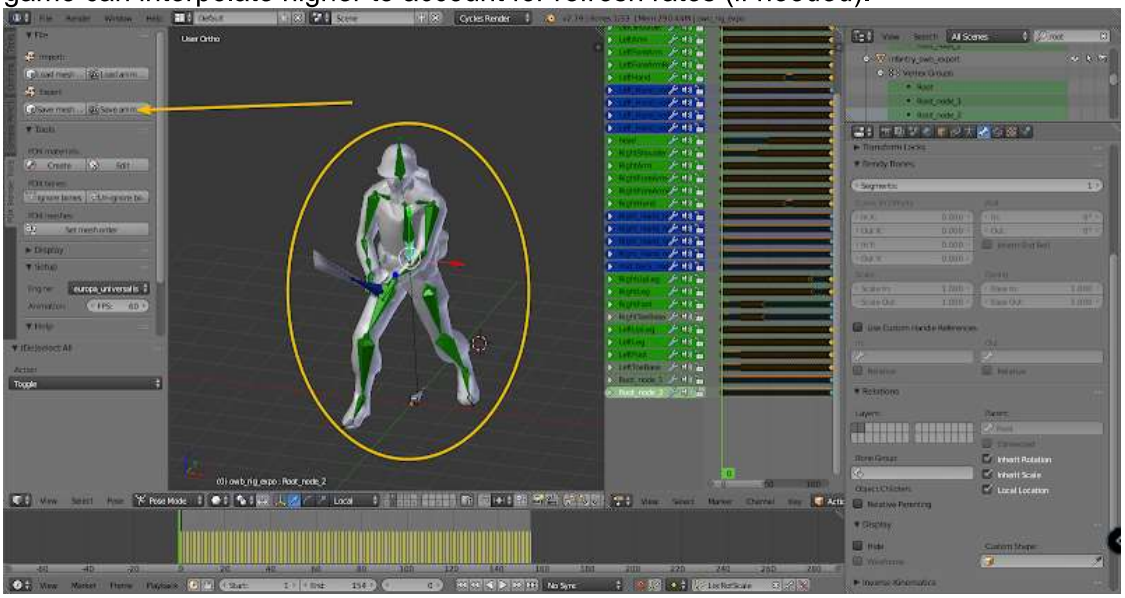
For exporting meshes with armatures, make certain that you have the mesh parented to the armature, the armature modifier is on the mesh with the armature as it's target, and is using vertex weights as the option. I prefer to have my root bone facing backwards, but as long as your mesh is facing Y- in neutral pose, it does not matter.

Export Animations

Animations are also quite easy to export. The two most important things here are to make sure that your rig has no transformations on it, and that your keyframes are baked.



Now we can go to our rig with only deform bones without constraints to export.
Select the armature in either pose or object mode. Make sure to set your timeline to the animation length, minus one frame, so you don't get a loop-seam (caused by your first and last keyframes being the same, as you have 2 frames the exact same). Press save animation and boom! It just works! Make sure to double check your animation speed (it defaults to 15 fps). I prefer 60 fps, as it makes very smooth animations. The game can interpolate higher to account for refresh rates (if needed).



Part 5: The Hook - Up

The .mesh File

The texture locations are only read from the same directory as the .mesh itself!

The .gfx File

The .gfx is what gives your little model life in the paradox database of model things. Here we declare: animations, shader, textures, .mesh, and scale. It is also used for some other things not worth mentioning as

they never appear relevant.

```
objectTypes = {
   pdxmesh = {
            name = "protectron_mesh"
            file = "gfx/models/OWBEntity/robots/protectron.mesh"
            scale = 0.45

            animation = { id = "idle"
                   type = "protectron_idle_anim" }
            animation = { id = "idle2"
                   type = "protectron_idle_2_anim" }
            animation = { id = "attack"
                   type = "protectron_attack_anim" }
            animation = { id = "move"
                   type = "protectron_walk_anim" }
            animation = { id = "retreat"
           type = "protectron_retreat_anim" }


            meshsettings = {
                    name = "jorodoxShape"
                    index = 0
                    texture_diffuse = "protectron_diffuse.dds"
                    texture_normal = "protectron_normal.dds"
                    texture_specular = "protectron_specular.dds"
                    shader = PdxMeshAdvanced
            }
   }
}
```

So lets step through this:
First you have the header for a .gfx file

```
objectTypes = {
```

We always need this at the top, closed at the bottom

```
pdxmesh = {
            name = "protectron_mesh"
            file = "gfx/models/OWBEntity/robots/protectron.mesh"
            scale = 0.45
```

Here we have our object type, a pdx mesh, the internal name of this GFX object, the ABSOLUTE file path from your .mod directory (in my case oldworldblues) and the scale.

```
 animation = { id = "idle"                    type = "protectron_idle_anim" }
            animation = { id = "idle2"
                   type = "protectron_idle_2_anim" }
            animation = { id = "attack"
                   type = "protectron_attack_anim" }
```

Next we declare animations for this mesh. There are two parts, the ID and the type. ID is what is referenced in the entity block inside a .asset file to actually play animations. Type is the internal register of the animation, which is also declared in a .asset file. It is important to keep these two different, with the ID being general, and the type being a unique name. For example, all of my models use the same set of ID's so I know exactly which animation is for what, at each frequency, but my type is always model_animation_anim.

```
meshsettings = {
                    name = "jorodoxShape"
                    index = 0
                    texture_diffuse = "protectron_diffuse.dds"
```

```
            texture_normal = "protectron_normal.dds"
            texture_specular = "protectron_specular.dds"
            shader = PdxMeshAdvanced
        }
```

These are the settings for the mesh found in the .mesh, which you can edit here! Important notes: the name should never change if exported with jorodox tools, the index is which mesh in the export it is, and is only important if you are exporting multiple meshes to one file. This is where the name and index you set earlier is important! If only one object in the file, you can omit these two lines. The texture declarations can take any file path, but important note: it is RELATIVE to the .mesh file path we declared earlier. For example: our .mesh is under mod/gfx/models/mymodel/mesh.mesh, our textures could be stated as /textures/texture.diffuse if we put them inside of mod/gfx/models/mymodel/textures/

The .asset File: Animations

This is the first introduction into the monster that is a .asset file. First we will touch animations, because this is needed for the .gfx.

There is no header on the .asset file type

```
animation = {
    name = "protectron_idle_anim"
    file = "robots/protectron_idle_anim.anim"
}
```

This is the animation block. We have the name, which is referenced as type in the .gfx (remember that?) and the file. The path is RELATIVE to the .asset file you declare animations in.

The Entity

This is the most complex part of adding a fully custom model into any paradox game. It handles animations, actions, mesh, attachments, the works. I'll break down every part I can, the best I can, but this file's parts are specific to HOI4. Animation states may not work in other games, but the general file will. To illustrate this: I have my power armor model in Stellairs, with animations:

Onto our entity's full code:

```
entity = {
    ###Name - entity name, structed: unitname_entity
```

```
    name = "power_armour_entity"
    ####paradox mesh - inside a .gfx file (powerarmor_meshes.gfx)
    pdxmesh = "powerarmor_t51_mesh"

    ####Animations and triggers

    ###state to be in when nothing else is triggered
    default_state = "idle"
    ####state to use when attacking an enemy
    ###second attack state (shooting gun)
    state = { name = "attack"                    animation
= "attack" animation_blend_time = 0.0  animation_speed = 1.0  looping
= no chance = 3 next_state = "attack" propagate_state = { rifle1 = "idle" } }
    state = { name = "attack"                    animation
= "shoot" animation_blend_time = 0.0  animation_speed = 1.0 looping
= no chance = 1 next_state = "attack" }
    ###defensive state
    state = { name = "defend"                    animation
= "shoot" animation_blend_time = 0.0  animation_speed = 1.0 }
    ####supporting another attacking unit
    state = { name = "support_attack"    animation
= "shoot" animation_blend_time = 0.0  animation_speed = 1.0 }
    ###movement on map with no enemies
    state = { name = "move"                          animation = "move"
 animation_blend_time = 0.0 animation_speed = 1.0 propagate_state =
{ rifle1 = "idle" } }
    ###retreat "limp"
    state = { name = "retreat"            animation="retreat" }
    #### death state (despawns after this)
    state = { name = "death"    animation="retreat"    }

    state = { name = "idle" animation="idle1" chance = 6 propagate_state =
{ rifle1 = "idle" } }
    state = { name = "idle" animation ="idle2" chance
= 2 next_state="idle1" propagate_state = { rifle1 = "idle" } }
    ### training state, normally many of these
    state = { name = "training" animation="idle1" chance = 6 propagate_state =
{ rifle1 = "idle" } }
    state = { name = "training" animation ="idle2" chance
= 2 next_state="idle1" propagate_state = { rifle1 = "idle" } }


    attach = {     name = "rifle1"    Right_Hand_Node = "minigun_pa_entity" }

    #attach = {     name = "rifle1"    hand_r = "minigun_pa_entity" }
    #attach = {     name = "rifle2"    hand_l = "laser_rifle_entity" }
    #attach = {     name = "rifle2"    hand_l = "minigun_pa_entity" }
    #attach = {     name = "rifle3"    Root = "laser_rifle_entity" }

}
```

This file is already commented, but I will explain the core functions anyways:

Name:

The name is described as such:
Identifier_(subunit type)_(visual level)_entity_(variant number)
The identifier is either a graphical culture (declared in graphical cultures .txt and stated in the country file for hoi4) or a country tag for hoi4, Ex. GER for Germany.
Subunit type is defined in your unit file. You can also use sprite types here (heavy, medium, light tanks). For aircraft you can use equipment names
Visual level is tied to the hoi4 equipment level, and declared in each level of the equipment. I only use one level, so I omit the visual level, as level 1 (starts at 1) does not need a number, but level 2 would need

xxx_2_entity.
Variant number lets you give some more flair. For example adding more entities for different camo options for your soldiers to use.

Stellaris follows a similar format, but after the graphical culture (identifier), the part name is listed as seen inside of the ship file you are making. EX mammalian_battleship_bow_04_entity.

Pdx_mesh:

This is the name we declared in the .gfx file we make earlier!

Clone:

```
clone = "light_robot_entity"
```

This option inherits all of the code for whichever entity is cloned, allowing any information inside the new entity to overwrite it. This is useful for when you need to retexture a unit for a different country, but want to keep the animations and not copy and paste the giant block of code sixty times.

Animations Anatomy:

Animations are made using “states.” These are hard defined in the game you are modding, so some research might be required. You can make custom states, and I will explain how to do that in a subpart of this section.

First we have the default_state. This is whatever the entity should spawn in as any time it is loaded up, be it from zooming, map view, spawning, whatever.

```
    state = { name = "attack"                    animation
= "attack" animation_blend_time = 0.0  animation_speed = 1.0  looping
= no chance = 3 next_state = "attack" propagate_state = { rifle1 = "idle" } }
```

So this is a state. We have a few components of it

Name - this is hard coded for the most part, but we can link custom states from a hardcoded one.
Animation - the animation ID we defined in our .gfx
Animation blend time - this gives a time in seconds to blend the animations of the previous state and this state.
Animation speed - decimal speed of the animation
Looping - this will loop the animation, but most states are continuous so this option may not matter, fiddling with it is the best bet
Chance - this allows random animations for a given state. For example here: we have a running animation, that occasionally fires a gun, as opposed to just standing still shelling the unit.
Next state - the state to move to after this state.
Propagate state - this forces any attachment to transition into the given state. For example above, we force whatever entity is attached to rifle1 to become idle, in this case, it forces the minigun to stop firing, or it may continue the next animation cycle.

Event: this is a declaration that does something at a specified time in a state.

```
event = { time = 1.6 node = "muzzle"  particle
= "heavy_tank_attack_barrel_particle" light = "mg_muzzle_flash" keep_particle
= yes sound = { soundeffect = heavy_armour_fire  }  }
```

Time - time in seconds to do something
Node - this is a locator object, either placed in an editor (blender or maya) or manually made
Particle - particle effect to spawn at the node
Light - light to spawn at the node (lights up other objects on the map)
Keep_particle - keeps the particle when the state is over, EX shells, fire, etc.
Sound - self explanatory
Entity - spawn a given entity at this time. Keep particle will despawn it if keep_particle = no

Attaching Entities:

Entities can be attached to other entities in two ways depending on the game. First is through locators. These are placed in maya with a locator object, or coded in by hand as such:

```
locator = { name = muzzle position = { 0 0 0 } rotation = { 0 0 0 } }
```

This is pretty self explanatory. Position and rotation are given as three Euclidean transformation based on the object's local axis. I'm never quite sure which is which, as exporting from blender uses a different coordinate system, but I am relatively sure X is right left, Y is up down, Z is "forwards" and "backwards". Rotation is given in degrees on the same axis. This will gimbal lock if you are doing weird stuff with it, so you may need to use two entities attached with the method below. We are basically inputting a transformation matrix from the object's origin, therefore it is important to know where the origin is!

Attaching is the other method and uses the joints themselves.

```
attach = {      name = "rifle1"     Left_Hand_node = "protectron_laser_gun_l" }
```

Name - arbitrary to what you need
XXXX = This is the bone's name, in this case Left_Hand_node is a bone in my rig inside blender
"Xxx" this is the entity you wish to attach.
There is unfortunately no rotation for this, so your bone must be the right way around!

Custom States and The Magic of Propagation:

One can't force an entity into a custom state of action so to say. Custom states allow one to make a chain of actions. For example, if you want to make a gun in stellaris, charge, fire, recoil and return, it is easier to use 3 states than one long animation and a bunch of events. Why? Because each animation and timing can be changed individually, allow with the fact that you can add CHANCE. Chance allows variation in the progression of an animation chain, so as to spice up the visuals more than you could otherwise.
Custom states are made like normal states, but you MUST use next_state = "custom_state" to trigger them, where hardcoded states will just trigger.

The magic of propagation and custom states is used in the protectron to make it fire fluidly in Old World Blues. I use an empty mesh with a location, attached to the hand. This then has a custom state I propagate when I need a laser particle. It triggers my laser object, which spawns another entity at the hand, that is set up to make a laser particle work properly.

Your Custom Model now works in-game! Congratulations

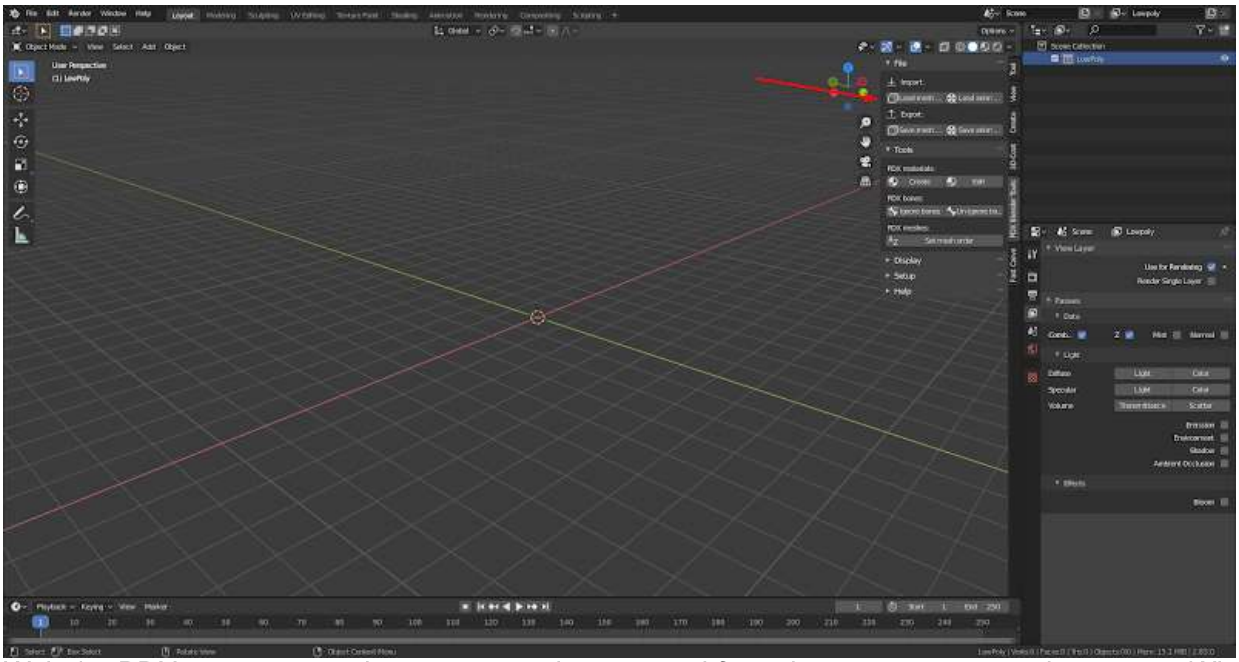
Part 6: Particles

Adding and editing particles (specifically in HOI4 and Stellaris) is fairly easy once you understand vaguely how the system works. The most important tool to use in this effort is the PDX Particle Editor. This comes bundled with HOI4 and Stellaris, I don't own EU4 or CKII so I can't test those. To open the editor, set launch options to -editor ONLY. On start, it will open full screen and probably not respond. Give it time, it has to load all of the graphics for you to be able to edit them. In this editor, you can call entities, meshes, and particles. Meshes you can check textures and such, entities you can check particles, animations, and states. Particles allow you to completely change and save new particles.

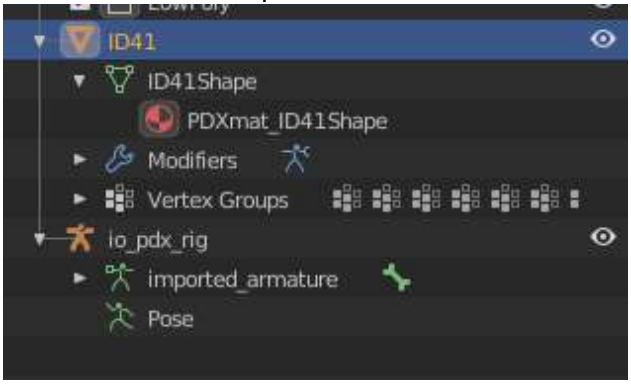
Particle Asset files

Part 7: Misc

Importing a Vanilla .mesh file



With the PDX io plugin, meshes can easily be imported from the same menu used to export. When imported, meshes have a few parts:

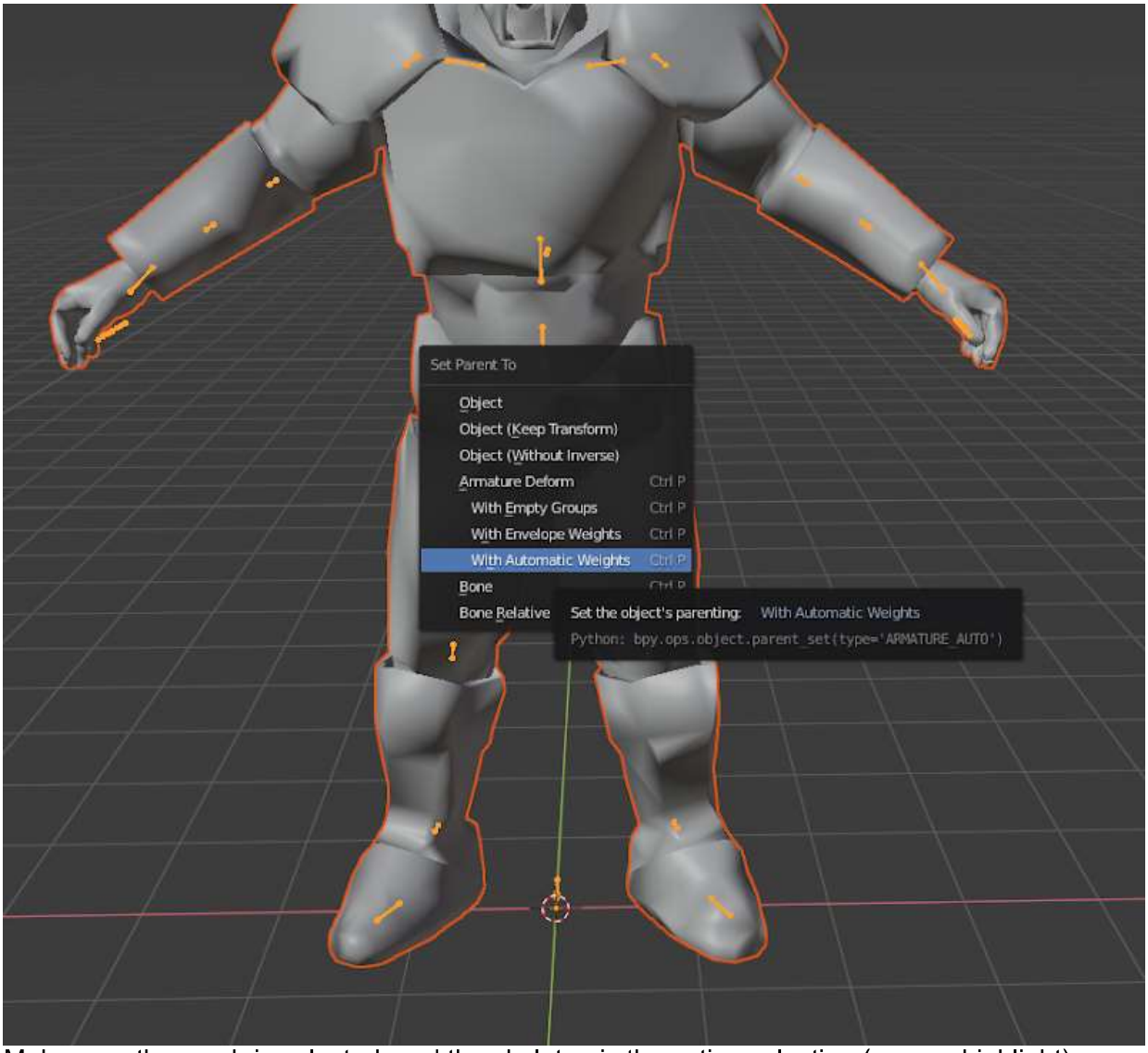


Each mesh comes with a shader that has the data for Clauswitz stored inside. It also includes the mesh with vertex groups/weights and the rig.

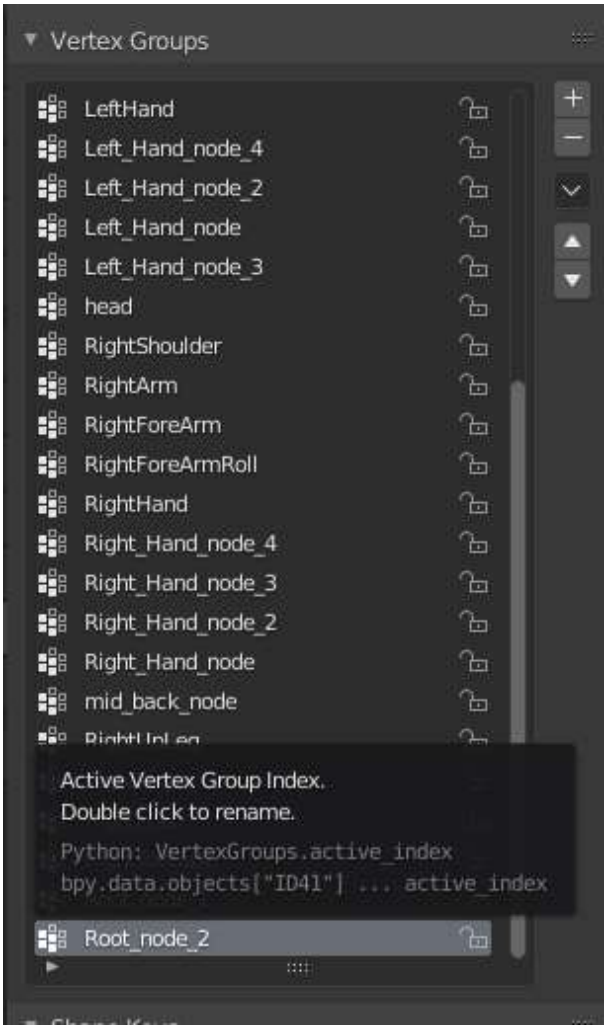
Exporting a new mesh onto the Vanilla skeleton

Before we begin, make sure that the mesh is clean of any errors mentioned in the beginning of the guide. Check for manifold geometry (little or no holes), triangulated geometry (or all quads), and no perpendicular-non manifold faces.

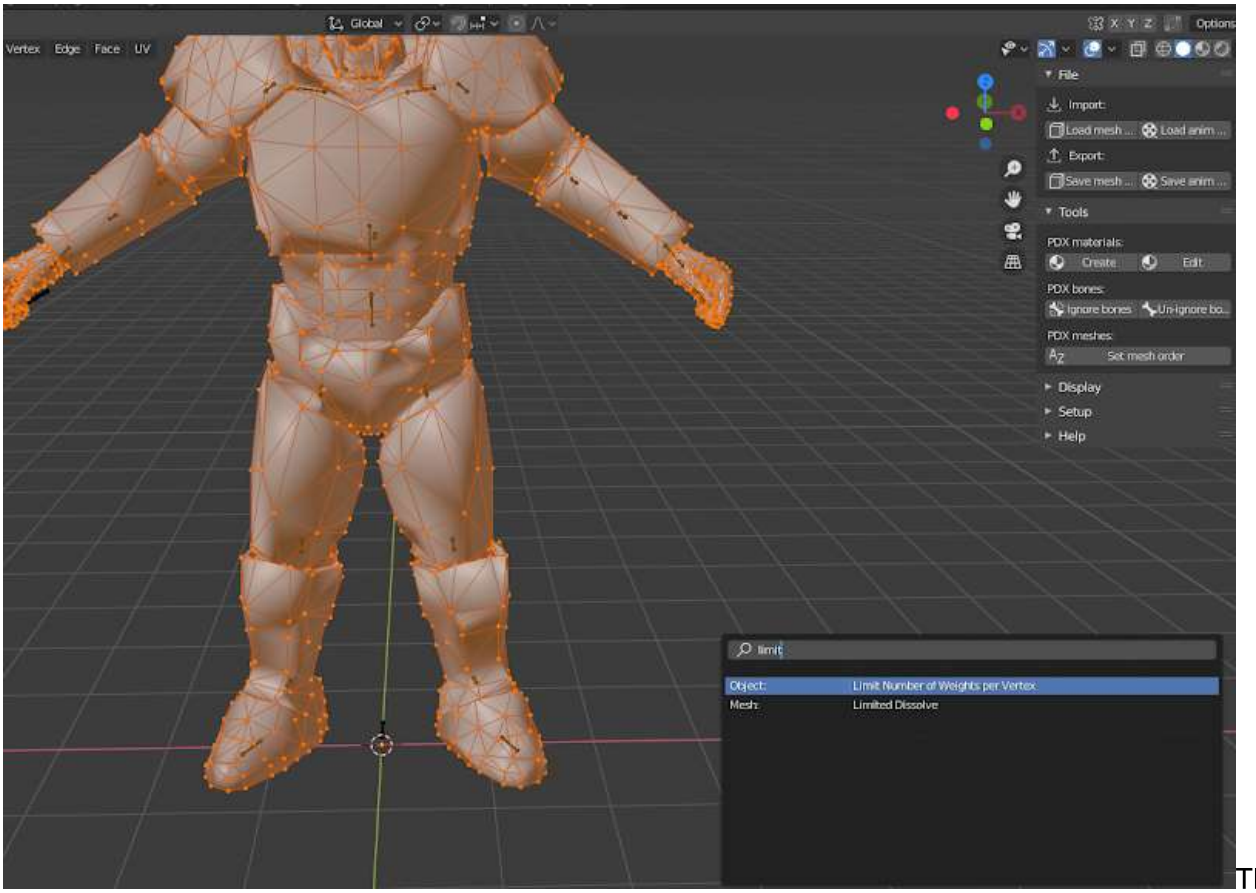
To export a new mesh onto the vanilla skeleton, first the new mesh object must be parented to the now meshless skeleton.



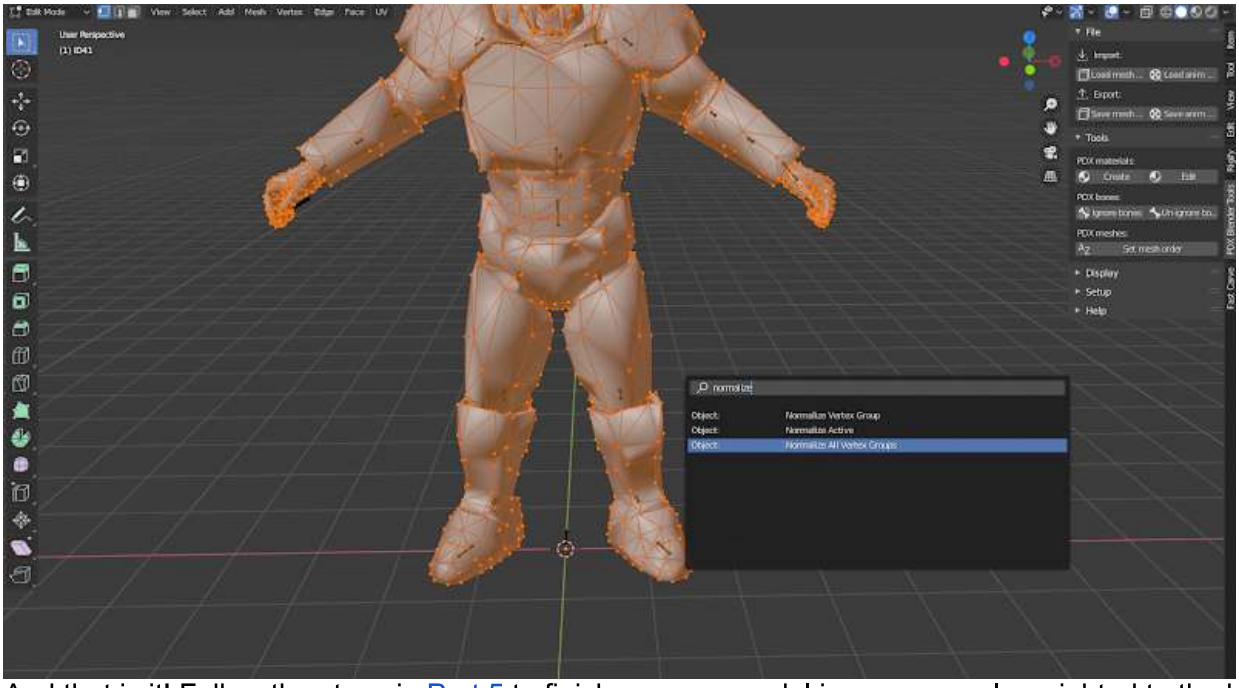
Make sure the mesh is selected, and the skeleton is the active selection (orange highlight).



This is the result, creating a vertex group for each bone on our rig. Next we want to select the rig and tab into edit mode



Then, we want to limit the amount of influences on any vertex to 3. To do this, select all with A, then press space and search for limit. Select Limit number of weights per vertex. When selected, change the setting to 3. Keeping the mesh selected, we then want to normalize all vertex groups, so the weight on a vertex adds to 1.



And that is it! Follow the steps in [Part 5]() to finish up, your model is now properly weighted to the base game mesh.