Engine22 Mapping – Object Modelling

1. Background

2. What is an "Object"?

When it comes to geometry in our worlds, we can roughly divide into two categories:

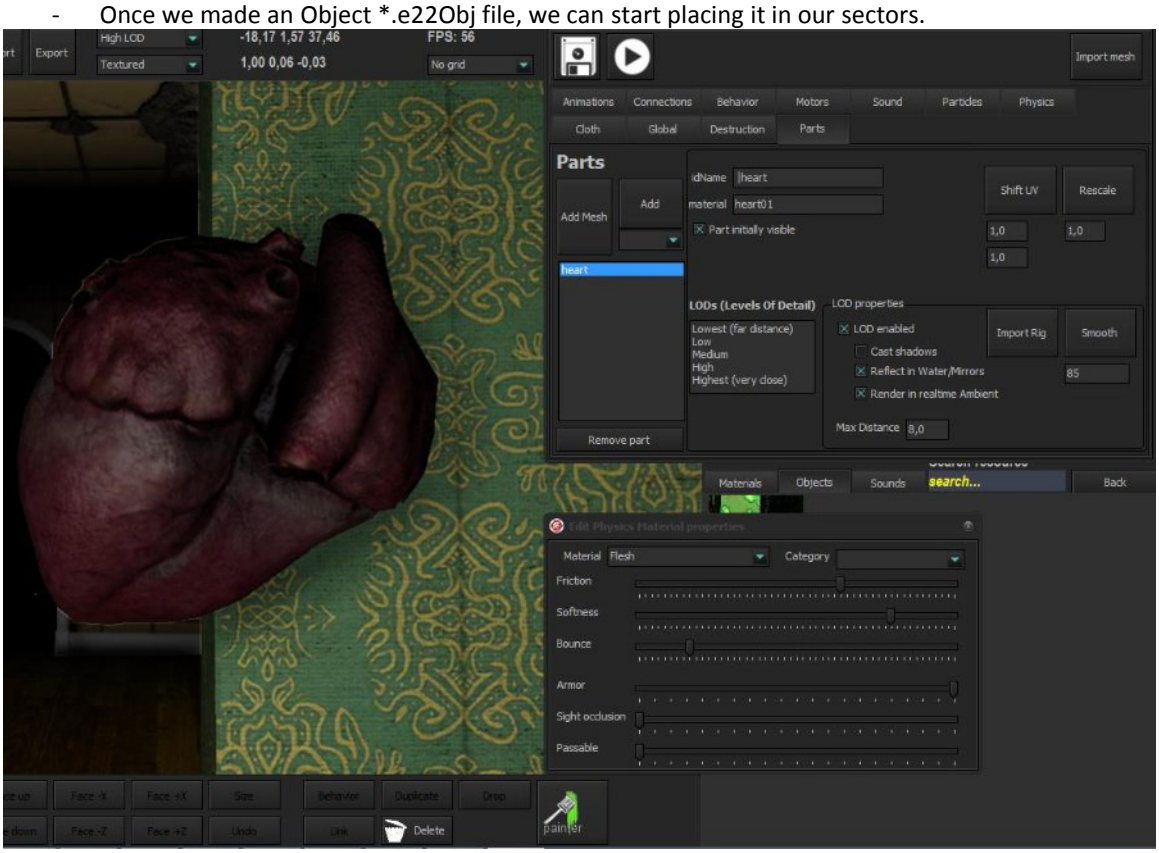
- Static geometry → Sector "chunks" such as walls, floors, stairs, pillars, …
- Dynamic geometry → Objects (or "Props") such as monsters, barrels, furniture, junk, …

Static world geometry is usually "one of a kind". A particular wall will only appear once with that exact position / shape / UV / material / … in the world. But we also decorate our worlds with objects that can appear more than once at different positions, and/or are meant to "move". With "move" we mean:

- Fall / collide / get bumped by physics → Crates, barrels, debris, …
- Can rotate / shift → Doors, windows, …
- Can walk or move by itself → NPC's, cars, the player, …
- Can animate (skeleton, joints) → Humans, Robot arm, Machinery, …
- Can be collected → Weapons, bonus items, health packs, …
- Can be destroyed → Gas-tank, breakable object, …

So, if we have an entity that either "moves", and/or is expected to be used multiple times, we usually decide to make it an "Object" instead. That means:

- We don't model objects into our Static Geometry (we can, but just don't export it, or name it "dummy")
- We model objects in separate files, and import them via MapEd's "Object Editor"
- Once we made an Object *.e22Obj file, we can start placing it in our sectors.



3. Object properties

Objects are not just decorative pieces of geometry with a certain material. A lot more can be done with them. Here an overview of object properties:

Global properties	**idName**	The name of this object, equal to the e22Obj filename. This is used in other files (such as sectors) to refer to this object, and has to be unique!
	In-game name / description	For custom game usage. Showing unit names, inventory descriptions, et cetera.
	Is Global	Local objects are bound to a sector, and will "reset" their state/position when you re-visit. Global objects will store their position/state into a different file permanently. Monsters, collectable items, puzzle objects or permanently destroyable objects are examples of this.
	Is Static	Static objects are supposed to stay stationary. They can't be moved by physics, and will be baked into IBL probes for reflections / GI.
	Thumbnail	A small picture of the object used in the Browser. Can also be used for inventories and such.
Parts		An object is made out of 1 or more "parts". Each part gets its own material and sub-LOD data.
	Material	The texture(s)/shader technique/parameters to be used. Can differ for each part.
	LODs	When rendering on a distance, objects can pick a lower-detailed mesh to gain speed. When drawing a jungle for example, we should use simplified low-poly models for the distant trees.
	Draw options	Render in reflections / GI steps / shadowMaps, et cetera.
Sub-entities		We can attach lightsources, sprites, particle generators, cloth, physic volumes (magnets, blowers, …) to an object.
Physics	**Collision hulls**	Each part can have 1 or more convex (point-cloud) hulls for collision detection.
	Hull properties	Each collision hull has physical properties so we know the friction, bounciness, what kind of sounds and particles to trigger on impact. It also has a "hitzone" name, which can be used in scripts or code to check special impacts. Use this to make weak spots on boss enemies for example.
	Overall mass	Heavier objects will be harder to push, and have a bigger impact when colliding on another.
	Buoyancy	How much it will float in liquids (0 means it sinks)
Motors		Basic movement such as sliders, shakers or rotators can be applied on our parts. For example, to make a fan rotate.
Animations	**Rig**	We import (FBX) files to rig our object, meaning we create a bone-hierarchy ("skeleton") and get the vertex-bone assignments ("weights").
	Animations	An imported (FBX) file that contains keyframes, telling which bones to move and where to.
	Events	We can insert events such as "play footstep sound", or "call code" at certain points during an animation.
Behavior	**Code**	LUA script, a Delphi class, or an external DLL can be attached to an object to "control" it. The code will intercept events such as "onHit", "onInterface", "onUpdate", and so on. Behavior code will be used to code A.I., but also to regulate more basic things such as what happens if a bullet hits.
	Properties	A list of custom properties used by your code. Each object-instance can alter these properties.

4. Object - Parts

As for modelling, the artist mainly has to care about the object parts and its different levels of detail.

Most objects are made out of a single part, but in case we have moveable sub-parts or want to use different materials, we'll have to split up in parts. For example:

- A door made out of a stationary frame & the rotatable door
- A human made out of a body & head
- A car with detachable parts (bumpers, cover, tires, windows)
- A frame with a rotating fan
- A desk with drawers that can slide in/out

For the sake of performance, try to avoid using more parts than necessary. Video-cards love to batch, so the less smaller objects and different materials to switch between, the better. Texturing / material wise, we can combine different sections into a single material, and use per pixel information to tell the glossiness, metalness, et cetera.

But if we have to split up, we can do that by naming our triangles properly in the (OBJ or LWO) model we'll import later on:

<meshType>.<partName>

high.Body high.DoorFrame
high.Head high.Door

Where "HIGH" stands for "high level of detail" by the way, see next chapter.

5. Object LODs (Level of Detail)

For each object-part, we can use multiple levels of detail. That means we use lower-poly variants for distant rendering. Up to 5 levels can be used:

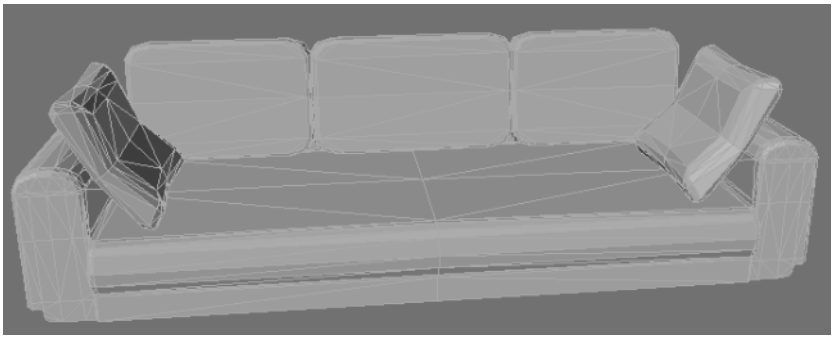
LOD	Usage	Typical range
Highest	For very nearby / higher detailed cutscenes	0 .. 2 meters
High	Nearby	2 .. 5
Medium	Somewhat distant	5 .. 17
Low	Background (small stuff may disappear here)	17 .. 40
Lowest	Far background (big stuff only, such as trees)	40 .. n

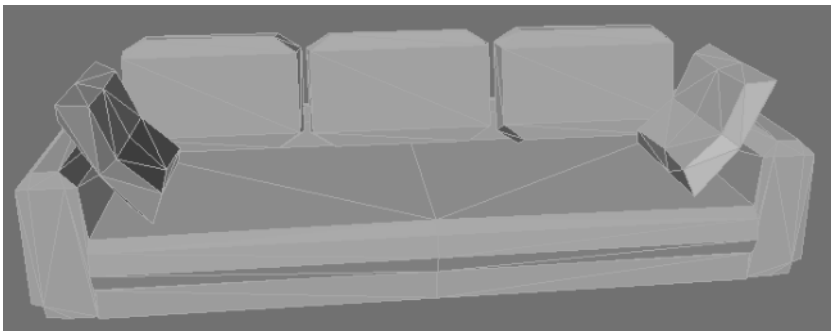
Note that we can change the distance-meters, and don't have to use all levels. Small stuff usually just disappears after X meters, so we don't define a Low(est) LOD. Or if our game just doesn't render anything in the distance, we can decide to use High(est) LODs only, and give them a large disappear distance:

Highest *distance 0..0* *(only use for custscenes, not in-game)*
High *distance 0..9999*
Med/Low/Lowest *not used*

Each object-part can define the LODs and distances. Within the model, we'll have to name the triangles as follow:

<lod>.<partName>
Highest.Monster
High.Monster
Med.Monster or Medium.Monster
Low.Monster
Lowest.Monster





6. Polycount?

Speaking of triangles, what is a good poly-count anyway? Well, that really depends. Is your target a modern computer, or a more limited platform? What is the development prognosis? If you're working on your game the next 4 years, expect the computers to become faster as well. And how about the other objects? You can easily render a single 100.000 poly model. But making a whole forest with 100k trees is not a very good idea…

There is no golden rule, but in general just don't use more polygons than needed. The high-res version of the sofa above uses 1400 triangles, which is fair for a game like Tower22 where you don't have thousands of objects in your view. T22 is also a slow-paced game, so you can really take time to examine objects from nearby, which requires some extra love & detail.

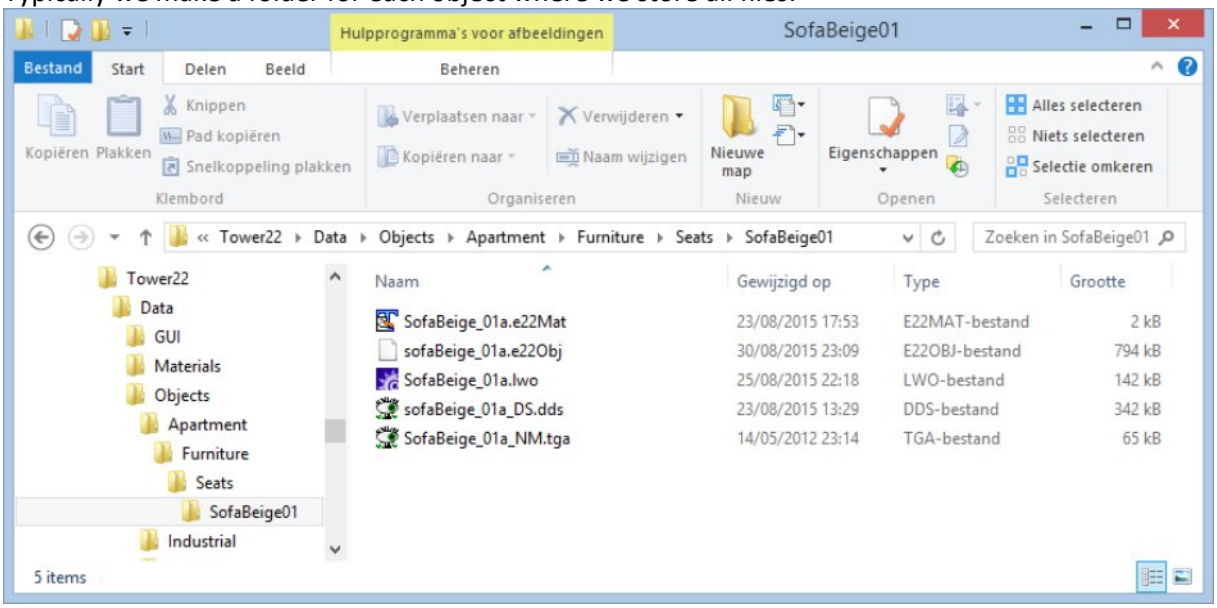
Most T22 objects have a polycount somewhere between 600 and 2.000. Smaller junk that will be used in larger quantities usually a polycount between 50 and 500. Monsters / players or first-person items (such as a gun) that have prominent roles, usually a count between 2.000 and 5.000. Probably we could use some more triangles, as the counts have never been a bottleneck so far. Then again, if it's not really needed… A lot detail can be "faked" with normalMaps, and Engine22 also supports Tessellation shaders eventually.

7. Naming & Materials

Naming is usually done by making a material with that name, and assigning the triangles to that material. The same material can also refer to a (diffuse/base)image file, for example "redBarrel01.dds". When importing, MapEd will try to find an equally named material for each part.

8. Files

Typically we make a folder for each object where we store all files:



Here we have:

- .E22Mat file → The material(s) used by this object
- .E22Obj → The actual object file, produced by MapEd or another Object Editor
- .LWO → Original "work" file. Lightwave or OBJ – you don't need this file later on
- .dds / tga / bmp → Image files used by Material. Preferably DDS.

Note that material, texture & object names need to be unique. So its wise to make the name "complete". Instead of just "sofa", we also involved the color and variant (01A) in the name.

Also note that materials can refer to other images in another file, used by another object for example.

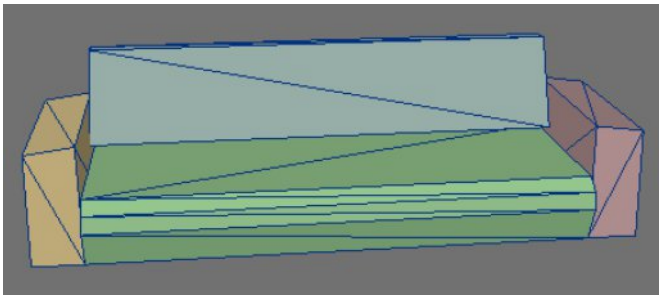
9. Physics Collision Hulls

Asides the actual parts/lods geometry, we also store “collision hulls” inside our files. These are named:

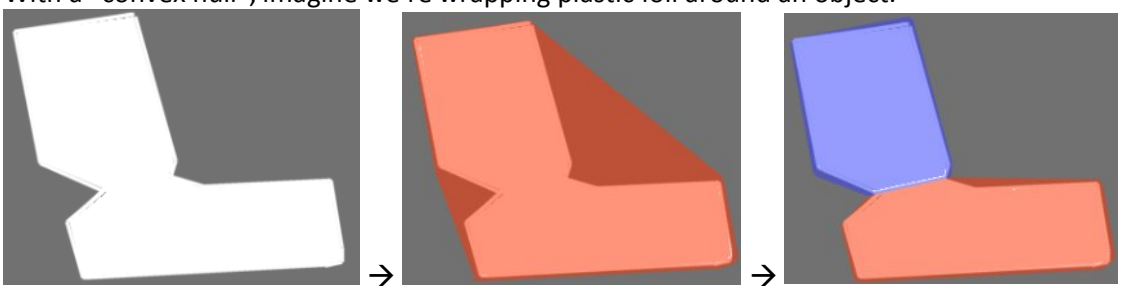
Hull.<name>
Hull.Torso
Hull.Head
Hull.LeftLowerArm

Hulls are usually simplified versions of the geometry. For example boxes that cover the shape of your object roughly. Each hull can have a different material (metal, rubber, flesh, …) for different physics characteristics and impact effects. And when a part of the the object collided, gets interfaced with (clicked, looked-at, use-button pressed, …), or gets shot, our (script) code will receive the given idName so we can identify what exactly got hit/interfaced with.

Since the hulls have to be convex, its often necessary to make multiple hulls as well. The sofa below used four.



With a “convex hull”, imagine we’re wrapping plastic foil around an object:



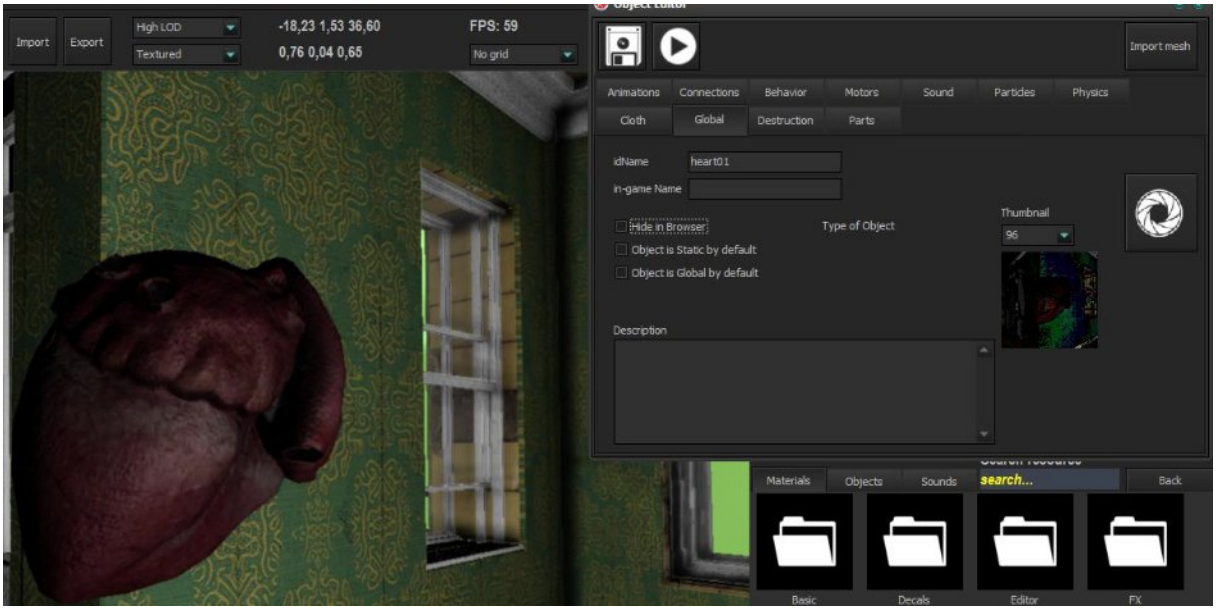
An object → 1 hull gets wrapped… not a good fit → 2 hulls, better fit.

10. Importing into MapEd

Click the “Object Editor” button, and then click “Import mesh”. It’s also a good idea to move into a sector where you like to use this object as well, so you can directly preview how it looks.

Assuming that we named everything properly, we can put everything in a single OBJ or LWO file. And then we just fill in the remaining properties. At a minimum, we should:

- Take a thumbnail screenshot
- Ensure the mesh looks ok
- Ensure each part has a material
- Select a “Behavior” (the code that controls/manages the object)



When ready, click the “Save” button. Now you can find the object in the Browser. Double click an object here to insert it. Then use the “movement” buttons at the bottom of MapEd, or numpad/arrow keys to place the object. You can also click the “Drop” button to activate physics. The object will fall down, plus you can use the arrows/numpad to apply force on it. Be aware that you can knock over other objects as well though!