Mapping Guidelines
v1.3

Last updated: 12 June 06
by pospi - http://pospi.spadgos.com

This document contains general guidelines for experienced mappers on the extra components of Furious Steals maps. Please note that unrealEd **must** be run with the switch *–mod=furiousSteals* for these components to become available. The easiest way to do this is probably to run the editor batch file in *UT2004\furiousSteals*.

Pathnode Placing

Pathnodes are used for a couple of different things within the game and so you should be aware of them.

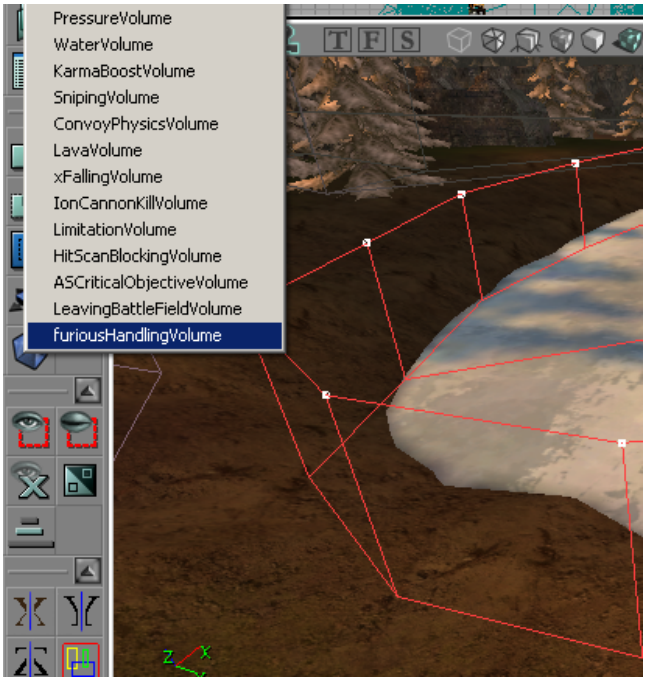
- Players will move to the nearest pathnode when they press their reset key, so avoid placing them in tight areas or unlikely places.
- The gold in Steals and Rabbit gamemodes will use a random pathnode as its spawn location. It is still ok to put pathnodes over water though as the gold will reset if it is submerged or in a pain causing volume. Again, don't put them in unreachable places or you'll end up halting the game.

Player Starts

Player starts work much the same way as in UT2004, except that their rotation is now irrelevant. Players will spawn facing the way their view is pointing. This is so that they will typically spawn facing the direction they died in and not lose any time via turning around.

Vehicle Handling Volumes

Furious Steals provides an extra volume type that you can use to modify the handling of vehicles. It extends from physicsVolume, so you can do anything with it that you could do with a physicsVolume as well.



Making a volume to control handling on this ice patch

Add it as you would any normal volume, by right clicking on the volume builder with a brush selected. The added properties it has are under the *furiousHandlingVolume* group, and are as follows:

vehicleLatFrictionMult
This is a multiplier for the sideways friction of the tires when inside the volume. Think of the default (1) as being a good amount for driving on a standard road surface like asphalt or concrete. Values lower than 1 will make cars slip around more, values greater than 1 will make them corner harder and behave like they're driving through glue or something. Note that this setting also has an effect on their lean, as higher values will make the tires grip more and thus the cars will have more of a tendency to flip over. Setting this to 0 or less will probably crash the game. Recommend about 0.2 for slippery areas - ice, oil etc; about 0.6 for grass and such. Play with to your taste, of course.

vehicleSpeedMult
Maximum speed and engine power of the vehicle will be multiplied by this amount when a vehicle is inside the volume. So obviously 1 is normal speed. Values greater than 1 will be interpreted as 1 (so you cannot make vehicles go faster than their normal speed), values of 0 or less will probably crash the game.

When using handling volumes, always try to give a clear indication as to when they will have an effect on the player. In the above screenshot for example, the volume is creating slip on an icy surface. It also certainly pays to keep things consistent, so that similar areas have similar settings on the same map.

Hideouts

Hideouts are the meat of the Steals gametype. You must place them down manually, and one at a time will be chosen based on a few criteria. Note that the same hideout will never be used twice in a row (unless there's only 1 of course).

To add a hideout, use the *Actor>furiousHideoutCore>furiousHideout*. **Never** place a *furiousHideoutCore* down or you will break things.

FuriousHideouts have a **bRandomiseRotation** property which will put them in a random rotation every time they spawn if true. Use this for hideouts out in the open to make them different every time. Definitely don't use it for hideouts in the middle of the street or you might find them spawning in very difficult ways.

Physics Objects

Furious Steals allows you to put cool dynamic objects in your levels that will work properly in multiplayer games. You can feel pretty free to put as many as you like in, because they reset themselves after a certain time to save on CPU resources. The exact time they take to reset can be configured by the game's host, and could be anywhere between 5 and 30 seconds depending on the speed of their computer and network connection. Point is, go nuts.

Because there are lots of properties to play with, we've included a fair number of preset objects for you to quickly and easily drop in your maps. We'll show you how to add those, how to build complex objects made up of smaller parts, and finally how to use your own staticMeshes to make your own fully dynamic environments.

General Guidelines

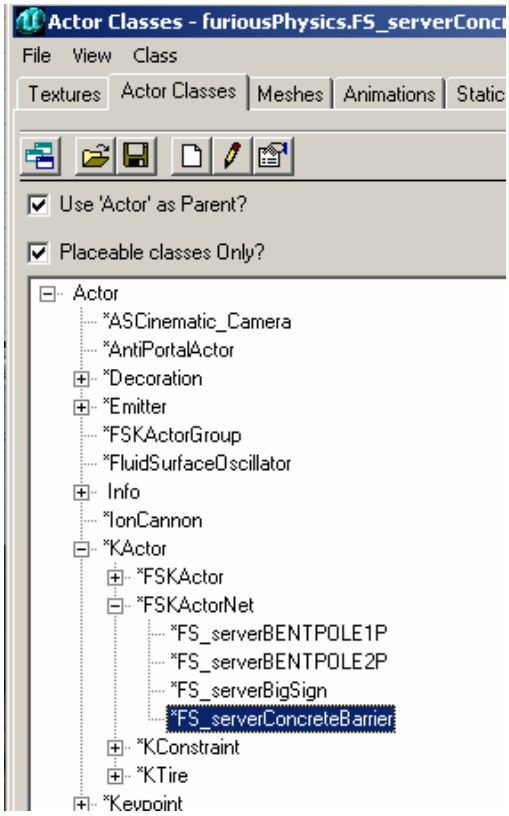
To save on bandwith even more, all our Karma objects start off disabled. This means that you shouldn't place them anywhere that would look weird, and expect gravity to put them in the right place for you. If you stick something up in the air, it will stay there until something runs into it. It will even go back there when it's reset.

Avoid going totally crazy with them and putting heaps in small areas. Try to spread them out a bit so that old ones have a chance to reset before the newer ones are disturbed. Remember that you still might totally lag the game if lots of physics objects are being simulated at once.

When building compound objects (see later on), remember that if any part of the object is bumped, the entire object will start moving. So again, don't use too many objects as part of a compound object because they will all be simulated together and the entire object won't reset until some time after the last piece is left alone.

Adding a simple object

This is a pretty easy thing to do, and you should have no trouble. The preset objects are under *Actor>KActor/FSKActorNeInterface/FSKActorNet/*, and all you have to do is select one, right click where you want to put it, and click 'add [whatever] here'. This path is actually contrary to the below diagram which is from an old build that I was too lazy to update. Sorry.



Where to find stuff.

Note that all these objects will snap to the grid, and their pivot points are at the base so you can usually just click on the ground where you want them to have them optimally placed.

Adding a compound object

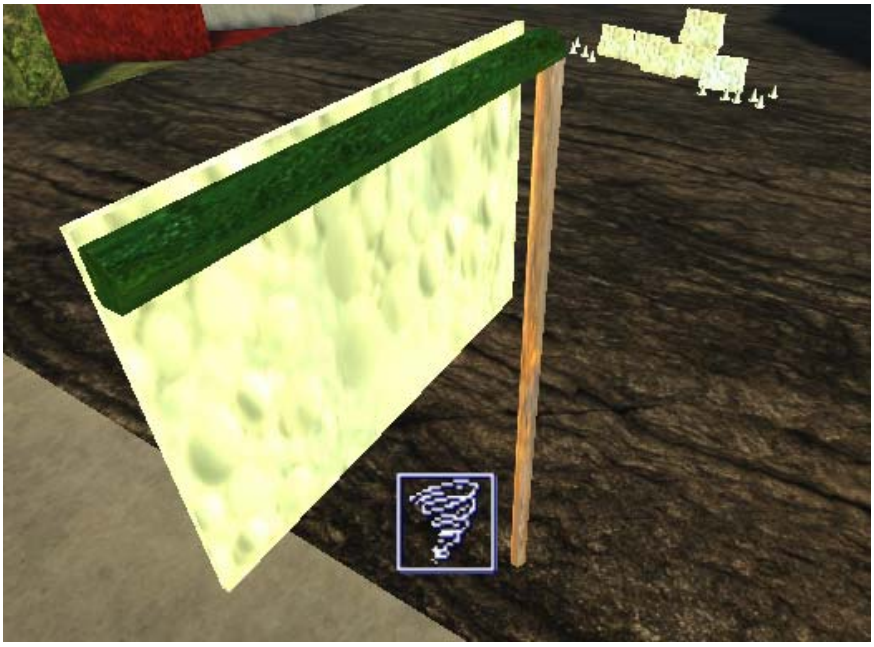
Adding a compound object is initially much the same deal as for simple objects. Our preset objects have been named to make things easier for you, for example *FS_serverBENTPOLE1P* and *FS_serverBENTPOLE2P* represent the two components of a bent pole. Anything named [whatever]#P is basically a component.

Our presets have their pivots placed in the right place to match up if you add them at the same location, so you can save yourself some hassle by turning your grid size way up and then adding the objects at the same grid position. There might also be other objects with their pivots set to match up, for example *FS_serverBigSign* and a couple of the traffic lights are set to match up with the bent pole pieces.

Feel free to make compound objects out of as many or few of whatever pieces you like – it doesn't matter if their pivots aren't in the same place, just move them to wherever you need them to be. Note that the component objects don't have to be touching, so you might be able to set up some cool traps and such.

Once you're done positioning your component objects, you need to add a master object to tie them all together. Add an *Actor/FSKActorGroup*

somewhere near your compound object so that you remember which group is referencing which objects.



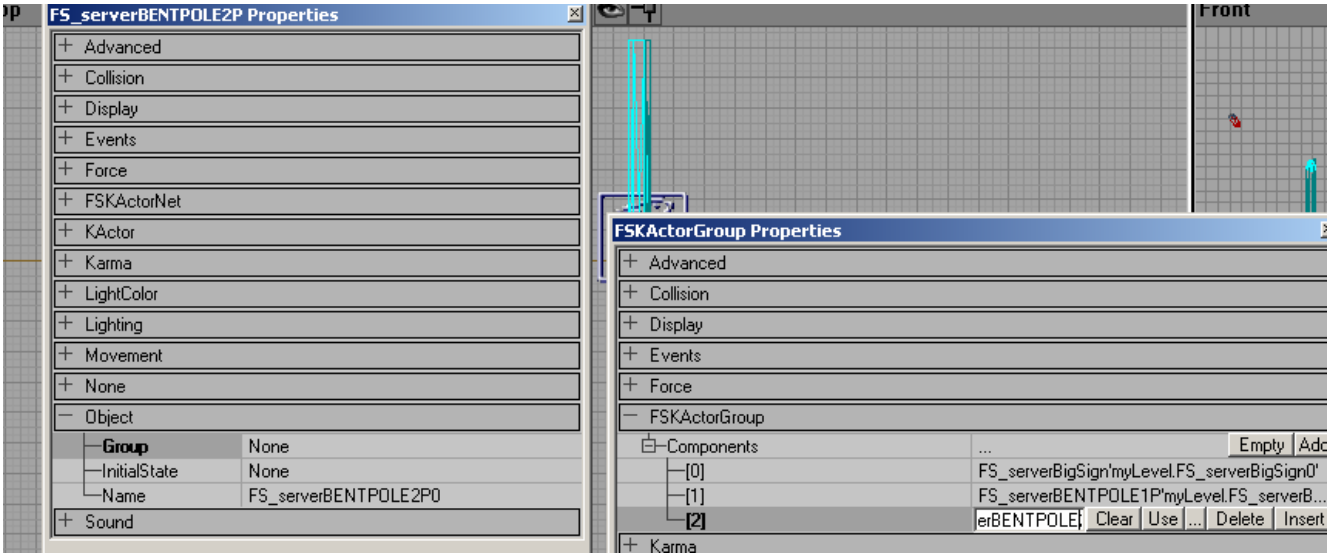
A sign made up of 3 components.

Now you need to tell the FSKActorGroup which objects to use. To do so, you need to input the names of all the component objects into it (note: when duplicating these groups the names should update for the duplicate, but it pays to check to be sure).

You could do this by opening the properties of each component object, going to *Object>Name*, closing that window, opening up the properties for the FSKActorGroup, going to *FSKActorGroup>Components* and typing the name. But that would be hard to remember.

I **really** recommend you download EditSelected, available here. It has a simple ut4mod installer and adds a button to the toolbar that looks like a couple of lists. Click it to open an object properties window that won't shut itself when you deselect the object.

Now you can have all your property windows open at once which makes typing in those names a lot easier.



Properties.

Click 'add' next to the *Components* array in the FSKActorGroup's properties and then type the name of each object in exactly as it appears in that object's properties box and press enter. If you got it right, it will change to show the full path for the object. If you got it wrong, it won't do anything or will possibly crash unrealEd.

Repeat as necessary for the other component objects, and you're done!

Making your own objects, and advanced properties

This is for people who are feeling advanced. Remember through this section that all the things I say you can fiddle with are ripe for fiddling with on the default objects as well if you want to change certain things about them. But don't be a dick and make silly things like cardboard boxes that weigh 100000 kilos and destroy vehicles instantly when they fall on them.

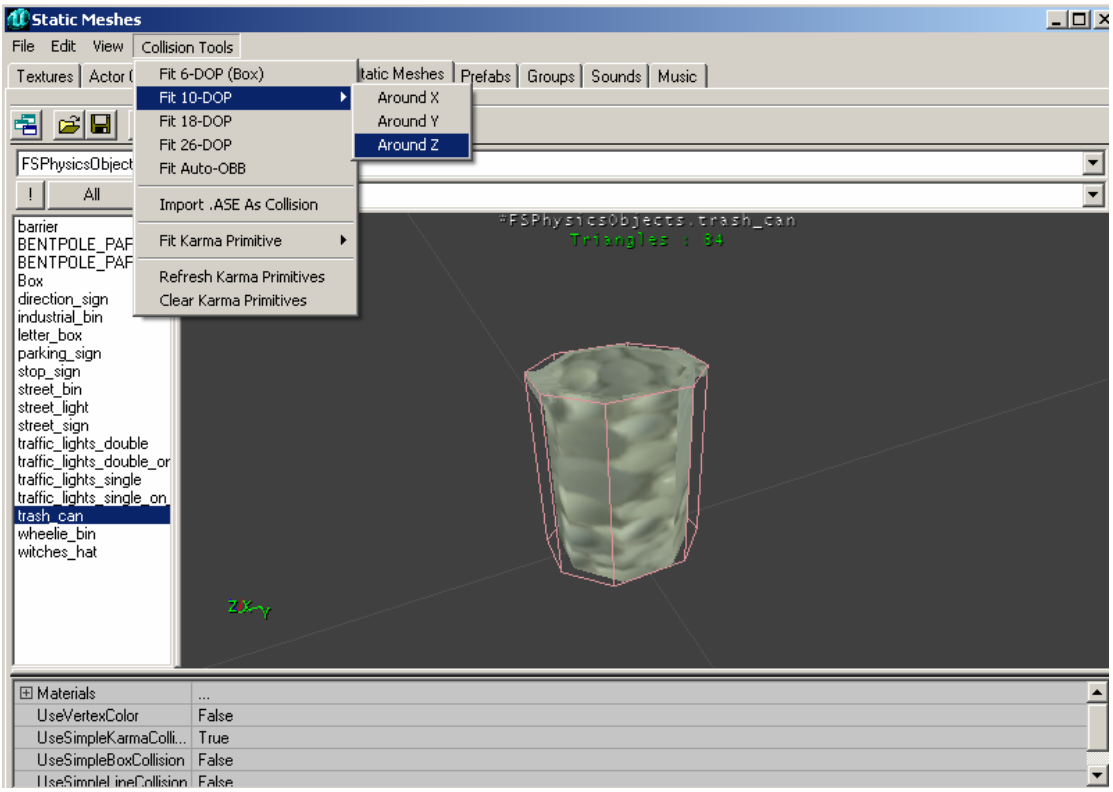
If you want to make a physics object, add an *Actor>KActor> FSKActorNetInterface>FSKActorNet*. This is the base class of all the physics objects that will sync up properly on all machines.

You can only use staticMeshes for physics objects, and those meshes must collide properly with karma (if they don't they'll just fall through the ground or not move at all). To make this happen, you either need to have imported your model with a collision model or you need to make your own in unrealEd. If you don't know how to import a collision model with your model then the latter is probably easiest for you. For the record, I'm pretty sure you can only model your own collision hulls in Maya.

Open up the staticMesh browser and find your staticMesh. (If you can't see it when you select it, go to the *View* menu, select *Auto Frame Selection* and reselect it.) Go to *View>Show Collision* so you can see what you're doing. The only way to make collision models in unrealEd is to use

the primitives in the first section of the *Collision Tools* menu. Experiment and find the one that fits closest.

If none of them fit very well, you could consider using full karma collision or splitting the object into subobjects. It will break into pieces with the second method, but that might be cool too. Full karma collision is potentially buggy, so check that vehicles don't clip right through it or get stuck in it or anything before you decide upon it (try turning down the object's KRestitution property in KParams to fix this if you can). To enable full collision, set *UseSimpleKarmaCollision* to false.



Doing stuff to staticMeshes.

Back to the physics object itself. The one you added before I got sidetracked will look like a box. Go into its properties because this is where we do basically everything.

Because it's easiest, I'll just list the properties and what they do. Anything not mentioned either shouldn't be messed with or probably has no effect. Change at your own risk.

Display

StaticMesh
the mesh to use for this object.

DrawScale
Multiply the size of the object by this number. DO NOT USE DRAWSCALE3D.

Skins
Array of materials to put on the mesh. If you want to reskin objects this is where to do it.

FSKActorNet

bBlockedPath
Used for bot pathing. Since bots are relatively unsupported anyway, should probably just leave it.

bCollisionDamage
Make sure this stays on.

bCriticalObject
This is to do with the server's options, as servers may disable all physics objects if they wish. If this is true, the physics object will stay in the level as a static object even if physics objects are disabled. If false, it will be deleted.

bRayTraceLighting
When true, the model will use the 'dramatic' lighting effect applied to players and stuff.

InitialImpactEffect
An effect to play when the object is first disturbed. Can be any actor as long as it destroys itself and has RemoteRole=ROLE_None. Basically *Emitters* or *xEmitters* only.

InitialEffectOffset
Offset relative to pivot point to play the initial effect at. This is in object coordinates (x=forward, y=left, z=up), and will take the rotation of the object into account as well. Your best bet is to measure this in your 3D modelling program as it will also scale by the object's DrawScale property and so mightn't be right in the editor if your drawScale isn't 1.

InitialEffectRotation
Same deal, but rotation relative to the object's rotation. Pitch=32767 is straight up fyi.

RespawnRadius
How far around the object to check for players before resetting.

KActor

bOrientImpactEffect
If true, the impact effects will be oriented to the angle of impact. Otherwise, they will all have the same (0,0,0) rotation.

ImpactEffect
Impact effect to play whenever the object is collided with. This effect will play at the place of contact.

ImpactInterval
Minimum time between impact events (sounds, particles etc) being called.

ImpactSounds
Sounds the object might play when hit. It will pick a random one to play each time.

ImpactVolume
How loud said sounds are. They'll fade off based on distance as well, obviously.

Karma

KParams

Karma parameters for this object. These are all the core parameters that control the behaviour of the object. I'll let this wiki page do most of the talking, although really most times you'll only want to play with the following:

KMass – sets how heavy the object is and how much damage it will do when it hits cars. 0.2 is quite light whilst 6 is pretty heavy. Note that the size of the object also effects its weight so this is more of a density value.
KFriction – how much the object will slide around and be pushable by vehicles. Lower KMass values also make it slide more since it's not being pushed down as much.
KRestitution – how 'springy' the objects is – 0 for solid and 1 for bouncy. If you're using thin objects leave at 0 or they'll get stuck in stuff.
KActorGravScale – how much gravity effects this actor. Can make it look unrealistic so only use if you really need to. I think if you set negatives the game will crash, but I'm not sure.

DO NOT CHANGE:
bHighDetailOnly (=false)
bKDoubleTickRate (=true)
bClientOnly (=false)
KStartAngVel (=0,0,0)
KStartLinVel (=0,0,0)
KStartEnabled (=false)
KImpactThreshold (=300)

Street Signs

Street signs are physics objects with some additional parameters. Basically, you can change the name of the street by a property if you want to be cool. This effect will only show in Direct3D, otherwise some blurry indiscernible text will be there instead.

Note that setting *Skins(0)* for this actor won't have any effect, use *signFallBack* instead.

bScriptStreetName can be set to false to disable the text if you want to use the object for another kind of sign or just put your own premade texture on it.

streetName
: The name of the street. The name should be in capitals, with lowercase 'st', 'rd' etc.

bScriptStreetName
: If false, no rendering of the street's name will happen and the material specified for '*signFallBack'* will be used regardless.

signFallBack
: Fallback material for players who can't see scriptedTextures. Also used if *bScriptStreetName* is false.

signBackground
: Background image to render under the text.

Traffic Lights

Just so that you are aware, the traffic light objects will pick one of two materialSequences for their skin. This is to mix up the lights so that not all are green at the same time. In any case, if you try to apply new skins onto the traffic lights, chances are they will be overridden.

Signs

Use the *FS_serverParkingSign* actor for square signs. There are a bunch of alternate textures in *furiousPhysicsObjects.Signs.** that you can apply instead of the parking sign texture via the actor's skins array. Feel free to make your own as well. The same deal applies to the big sign preset – plenty of extra textures to go around.