Jed DLL plug-ins

By Alexei Novikov

DLL plug-in interface is a new, faster and more elaborate plug-in interface for JED. However, there's a tradeoff - you'll need to know quite a bit of programming and use some "real" programming language to be be able to use it. Note that this interface is an addition to the old OLE interface, not the replacement. You can use the plug-ins of both types at the same time.

In order to be able to create this new type of plug-ins you need a programming language that can create Windows 95 DLLs (Dynamic Link Libraries) and support Microsoft COM (Common Object Model) interface. That includes C++ and Delphi 2.0+, at least. However at this time (version 0.91) only Delphi 2.0 and 3.0 interface units are provided. It shouldn't be too hard to translate them to C++ headers though. I hope to do it in later versions.

Unfortunately, I don't have time to explain the whole things in details, so here's some key point that hopefully should be enough to get you started.

The way the plug-in interface works is the following - you place a .DLL of a plugin to the JED's plug-in directory (along with optional .DSC file, which is the same as for OLE plug-ins) and it shows up in Plugins menu in JED. When you select the plug-in in the menu, JED loads the .DLL to memory, and invokes the function in your DLL named "JEDPluginLoad", passing a COM interface object as a parameter to it. You can use this object to access JED's data and functions, much like the TJEDApp object in OLE plug-ins. The function should return true if your plug-in worked successfully and false if some error occured (that'll just make JED report the error, you can report it yourself and just return true always).
A note - when you select plug-in in Plugins menu second, third, etc. time, JED uses the already loaded DLL (so your variable will have the values you left in them after your plug-in worked the first time). In some cases you might need to handle this situation in some special way.

The COM interface object passed to JEDPluginLoad function is the key to the entire interface with JED. Oops! I just realized something. I forgot to add "stdcall;" to the declaration of JEDPluginLoad function type, so JED calls it using Delphi call convention, passing parameters via registers. Ahem... Well, I can fix it in the next version by allowing to have an alternative function JEDPluginLoadNew() or something. For now it look like you'd have to use Delphi to make JED plug-ins. Or maybe C-style register call convention is compatible with it? Oh, well.

Anyway, that interface object passed to JEDPluginLoad is declared in the interface units provided as object IJED. Here's the quick description of function whose purpose/parameters are not too obvious.

```
  Function GetLevel:IJEDLevel;
```

This function return IJEDLevel ineterface object that you can use to access level data directly (much like TOLELevel obejct in OLE interface). Note that in Delphi 3.0 unit the return value is defined as pointer. That's because of some strange problem I've been getting with Delphi 3.0 when I defined it as IJEDLevel. So you'll need to typecast it to IJEDLevel when assigning to a variable. Like:
level:=IJEDLevel(jed.GetLevel);

```
  Procedure GetLevelHeader(var lh:TLevelHeader;flags:integer);
  Procedure SetLevelHeader(const lh:TLevelHeader;flags:integer);
  Procedure GetSector(sec:integer;var rec:TJEDSectorRec;flags:integer);
  [...]
```

The set of GetXXX/SetXXX procedures use the following conventions: the "flags" parameter is a set of bit flags which determine which field of a record passed to the procedure are to be retrieved/set. For instance, to get sector's sound and volume you would use:
level.GetSector(sec,secrec,s_sound or s_sndvol);
Note that after this call only secrec.sound and secrec.snd_vol fields of the record will be filled, the rest of them will not be modified. A note here: string values are returned as null-terminated strings (pchar). However, when you MUST not modify them directly, as JED returns the reference to its internal data. To modify a string field, copy a string, modify it, assign the pointer in the data structure to your new string and then set it with .SetXXX() call. I.e. something like this:

```
level.GetSector(0,secrec,s_sound);
StrCopy(newstr,secrec.sound);
{perform modifications of newstr}
secrec.sound:=newstr;
level.SetSector(0,secrec,s_sound);
```

The most valuable addition, probably, is a set of functions to perform editing operations, which are direct interface to functions used in JED itself. I hope most of them are self-explanatory. Here's the description of those that seem not to be:

```
  Function FindCollideBox(sec:integer;const bbox:TJEDBox;cx,cy,cz:double;var cbox:TJEDBox):boolean;
```

Finds sector's collide box. bbox is sector's bounding box (returned by FindBBox()).

```
  Procedure RotatePoint(ax1,ay1,az1,ax2,ay2,az2:double;angle:double;var x,y,z:double);
```

Rotates a point around an axis going from ax1,ay1,az1 to ax2,ay2,az2 by angle "angle".

```
IsPointOnSurface(sc,sf:integer;x,y,z:double):boolean;
```

Checks if a given point is within given surface. It assumes that the point lies of the surface's plane. Typically it's used in checking if a specific line intersects a surface. First you find and intersection of the line and surface's plane (LinePlaneInetsection() function) and then check if the point of intersection is within surface.

```
  Function CleaveSurface(sc,sf:integer; const cnormal:TJEDvector; cx,cy,cz:double):integer;
  Function CleaveSector(sec:integer; const cnormal:TJEDvector; cx,cy,cz:double):integer;
  Function CleaveEdge(sc,sf,ed:integer; const cnormal:TJEDvector; cx,cy,cz:double):boolean;
```

Performs regular JED's cleave operation of specified item. Returns -1 (or false) if no cleave was performed and the number of new sector/surface (or true) if it was. cnormal is the normal of the cleaving plane and cx,cy,cz is a point this plane passes through.

```
  Procedure CalculateDefaultUVNormals(sc,sf:integer; orgvx:integer; var un,vn:TJEDVector);
  Procedure CalcUVNormals(sc,sf:integer; var un,vn:TJEDVector);
  Procedure ArrangeTexture(sc,sf:integer; orgvx:integer; const un,vn:TJEDvector);
  Procedure ArrangeTextureBy(sc,sf:integer;const un,vn:TJEDvector;refx,refy,refz,refu,refv:double);
```

This is a group of procedure to manipulate texturing. For explanation of texturing normals, check out my second vector article. Er, sorry don't have a link handy. Orgvx parameter specifies a vertex to be used as a reference point. For an arbitrary reference point use ArrangeTextureBy() where refx,y,z and u,v specify a point and u,v coordinates in it.

```
  Procedure RemoveSurfaceReferences(sc,sf:integer);
  Procedure RemoveSectorReferences(sec:integer;surfs:boolean);ct;
```

Removes the reference to a gives sector/surface from COGs, adjoins and things. You should use these before you delete a sector/surface. Note that DeleteSector() and SectorDeleteSurface() functions do that for you. You only need to use these if you perform some fancy modifications of the level geometry yourself. "surfs" parameter of "RemoveSectorReferences" specifies if references to surfaces in the sector should also be removed.

Well, this decription is pretty sketchy, that's all I could do in such limited time. I'll update it some time.
Alex.