Dr. House – Developer Documentation

This application is a basic graphical user interface framework for experiments on architectural meshes. The meshes are displayed in the form of wires or polygons inside the window and can be modified using custom controls in the control box at the left border of the window. By the support of *OGRE3D*, arbitrary information concerning the displayed meshes can be overlaid at 3D coordinates. The mesh can be translated and rotated using the mouse and keyboard.

Microsoft .NET controls that provide custom test procedures can easily be added using the Visual Studio GUI.

The following documentation is based on a symmetry detector that is implemented within the Dr. House framework.

Contents

Developer Documentation

Basically, the program underlying *Dr. House* is split into two elementary parts: the core part and the customizable part. The former provides code that ideally doesn't need any adjustments, for instance OGRE initialization, navigation or mesh visualization procedures. In contrast, the customizable part is a skeleton for custom code that is controlled by custom user controls (please refer to the User Documentation) and operates on data provided by the core part as well as its own data.

The following sections inform about the project directories and outline the relevant classes of the core part, followed by a customization tutorial.

Directories and Files

Subversion repository: *https://svn.cg.tuwien.ac.at/data/svn/semhouse*

DrHouse main project directory

papers pertinent publications

DrHouse/doc documentation

DrHouse/*.cs complete C# source code

DrHouse/cpp custom C++ parts in the form of a self-contained DLL project called *dll* (the DLL contents are referenced by the main project's C# code)

ANN: Approximate Nearest Neighbor Search

FAMS: Fast Adaptive Mean Shift (efficient clustering algorithm)

DrHouse/bin binaries (DLL assemblies and executables)

DrHouse/skeleton configuration files and resources required during runtime

DrHouse/Properties some Visual Studio generated files

Files located in *DrHouse/skeleton* are automatically copied to the *Debug* or *Release* binary directory after the solution successfully compiled. In the project properties of the main project, the copy commands may be altered. Currently, the following commands are defined:

```
if $(ConfigurationName) == Debug copy "$(ProjectDir)\skeleton\mogre-release\*" "$
(TargetDir)"

if $(ConfigurationName) == Debug copy "$(ProjectDir)\skeleton\debug-config\*" "$
(TargetDir)"

if $(ConfigurationName) == Release copy "$(ProjectDir)\skeleton\mogre-release\*"
"$(TargetDir)"

if $(ConfigurationName) == Release copy "$(ProjectDir)\skeleton\release-config\*"
"$(TargetDir)"

copy "$(ProjectDir)\skeleton\scene\*" "$(TargetDir)"

copy "$(ProjectDir)\skeleton\settings\*" "$(TargetDir)"
```

Core Part Outline

The *Core* Object

(class *DrHouse.Core*)

Partial Class File	Contents
Core.cs	constructor, singleton instance, accessors
Core.Init.cs	initialization and mesh loading
Core.OutputUpdate.cs	update of the OGRE window
Core.Query.cs	auxiliary settings query procedures

The *Core* object is used as singleton. It is the program's object reference pool and furthermore provides vital initialization and data output methods. It can be regarded as the application's central object. The code of the *Core* object can be extended using the *CustomCore* class.

Especially the initialization sequence may be of particular interest: It is located in the *Core.init* method and invoked by *DrHouseForm.DrHouseForm_Load*.

Accessors of important objects the references of which are held by the *Core* object:

Form The application window (link to the application window's controls).

OgreWindow The panel that displays the OGRE output and receives navigation input from the mouse and keyboard.

Window, Camera, Viewport, SceneManager, *Node... OGRE's actual render window and further objects owned by OGRE.

ResultsForm A small control and log output window.

Mesh The loaded mesh.

****Vis*** (e.g. *MeshVertexVis*) mesh overlays (vertices, edges, selections).

Settings Link between controls, algorithms and hard disk.

The *Mesh* Object

(class *Geometry.Mesh*)

Provides the mesh's vertices in the form of *Geometry.Vertex* objects. They are queried from the internal *OGRE* mesh structure by the *GetMeshInformation* and *compileVertexArrayAndEdgeArray* procedures. A *Geometry.Vertex* object provides the following information:

index a unique identifier

position x, y and z coordinates

normal normal vector provided by *OGRE*

neighbors set of the neighboring vertices

OGRE-related information:

ogreMeshIndex identifies the underlying *OGRE* mesh (generated by the *compileVertexArrayAndEdgeArray* method)

ogreEdges list of edges that end at the vertex, provided in the form of *Mesh.Edge* objects that contain a reference to the *OGRE* edge, the end vertices of the edge and the normal vectors of the adjacent polygons.

The mesh object is stored in the *DrHouse.Core* singleton instance and can be accessed at using the *Mesh* accessor.

Settings and *CustomControl* Objects

(classes *DrHouse.Settings* and *DrHouse.CustomControl<T>*)

The *Settings* object is a central data base that stores all settings, basic settings as well as any custom settings. Since individual settings are usually manipulated by individual controls, the settings keys are named by the corresponding controls. For instance, the navigation speed setting is called *speedTrackBar*. Effectively, the settings object is a non-volatile control cache, the contents of which survive on the hard disk.

Currently, the Settings object can store and deliver data of the types *string*, *bool*, *int* and *float* (however internally, data is always *string*).

CustomControl is convenience wrapper for *System.Windows.Forms.Control*s that provides a built-in link to the *Settings* object as well as an event handler skeleton. The usage is described in the course of the custom part description (section *Custom Part*). Usage examples can be found in the partial class file *DrHouseForm.Controls.cs*, where *CustomControl* is used for the core part controls.

The *DefaultInputHandler* and *VertexPicking* Objects

(classes *DrHouse.DefaultInputHandler* and *DrHouse.VertexPicking*)

All input handling is processed by the *DefaultInputHandler* object. Except for the vertex picking handlers, which are added by *Core.init* (because the *VertexPicking* object is maintained by the *Core* object), all input handlers are added in the *DefaultInputHandler* constructor.

The *VertexPicking* object is responsible for vertex selection. To reduce processing time for large meshes, it makes use of *Core.VertexOctree*.

The *Progress* Class

(*Interaction.Progress*)

The *Progress* class is the interface to the *mainProgressBar* control (located below the *Open mesh...* button) and can also be used for other progress bars. It can be connected to a *System.ComponentModel.BackgroundWorker* object, which supports threaded execution. However, in any case, the progress should be reported using the *Progress.setFraction* and *Progress.setPercent* methods (which internally call *BackgroundWorker.ReportProgress*). If *mActiveBackgroundWorker* is set to *null*, the progress bar is directly modified by the

Progress object.

If a *BackgroundWorker* object is used, adhere to the following recommendations:

- The existence of the *THREADED* preprocessor determines whether the *BackgroundWorker*'s delegates are directly called or called by a separate thread.
- Increment the *BackgroundWorking* member of the *Core* object by 1 before a separate thread is started. After the thread finishes, decrement *BackgroundWorking* by 1. You can use this variable to test how many threads are currently active.
- Set the *ActiveProgress* member of the *Core* object to the *Progress* object if one exists (or otherwise set the *ActiveBackgroundWorker* object to the *BackgroundWorker* object), to indicate that the thread is cancellable. After the thread finishes, unset the *ActiveProgress* member by assigning *null*. This information is used by the *cancelButton_Click* event in *DrHouseForm.cs*.
- Set the *Form.cancelButton.ForeColor* to *Color.Red* before a separate thread is started and back to *Color.Teal* after the thread finishes.

The following code snippet shows the invocation of the initialization thread (file: *Core.OutputUpdate.cs*):

```
if(BackgroundWorking < 1)
{
    initBackgroundWorker_Prepare(Form.initBackgroundWorker, null);
#if THREADED
    Form.initBackgroundWorker.RunWorkerAsync();
#else
    initBackgroundWorker_DoWork(Form.initBackgroundWorker, null);
    initBackgroundWorker_ProgressChanged(Form.initBackgroundWorker, new
ProgressChangedEventArgs(100, null));
    initBackgroundWorker_RunWorkerCompleted(Form.initBackgroundWorker, null);
#endif
}
```

Except for *initBackgroundWorker_Prepare*, the *initBackgroundWorker** methods are added to the *BackgroundWorker* object by the *init* method (file: *Core.Init.cs*). The **_Prepare* method increments BackgroundWorking, **_RunWorkerCompleted* decrements it.

The *Octree* Class

(*Geometry.Octree*)

Provides an octree for vertex picking, but can be used for arbitrary purposes and extended if necessary.

Custom Part

Use the custom part to implement your application.

The skeleton of the custom part consists of the classes or files

- *DrHouse.CustomCore* (which is derived from *DrHouse.Core*)
- *DrHouseForm.Controls.Custom.cs*
- *DrHouse.CustomScene*
- *Geometry.CustomMesh* (which is derived from *Geometry.Mesh*)

Arbitrary additional classes may be added to the project in order to provide custom functionality.

The *CustomCore* class is meant for application-specific object references and initialization procedures. The *initCustom* method is called by the *Core.init* initialization sequence. Use the following methods for output update:

updateAllCustom(flags) This is the general update handler. Currently, flags will only be *0* (unspecific update) or *CORE_EVENT_VERTEX_SELECTED* (selection of a vertex happened). Usually, it simply performs a vertex/edge/... visualization update.

clearSelection turn selection off (afterwards, usually vertex/edge/... visualization is updated)

The connection between your custom controls and the settings data base is located in the *DrHouseForm.Controls.Custom.cs* file, the OGRE scene can be customized in *CustomScene.build* (add ground plane, lights, materials) and *CustomMesh* allows for application-specific mesh processing.

Very Important: The *Rerender* Flag

Set the *Core.RerenderRequired* flag to *true* in order to inform the renderer that something changed. Otherwise, changes will not immediately be rendered. It will automatically be reset when the rendering is done.

Important Hint: Use the *internal* access modifier for controls

In order to directly access controls of arbitrary forms from arbitrary classes, set the control's *Modifiers* to *Internal* (instead of *Private*) in the Designer of Visual Studio.

Custom Controls

CustomControl is convenience wrapper for *System.Windows.Forms.Control*s that provides a built-in link to the *Settings* object as well as an event handler skeleton. Usage examples can be found in the partial class file *DrHouseForm.Controls.cs*, where *CustomControl* is used for the core part controls.

Currently, the following types are supported:

Type	Controls
CustomControl<bool>	*CheckBox* and *RadioButton*
CustomControl<int>	*NumericUpDown* and *TrackBar*
CustomControl<float>	*NumericUpDown*
CustomControl<string>	*ComboBox* and *CheckedListBox*

If an event handler is required, for the sake of clarity a *CustomEventHandlerProc* procedure should be set when the *CustomControl* object is initialized and preferred over the Visual Studio Designer's double-click generated event handlers.

Usually, a *CustomControl* object is created in a one-liner, as demonstrated by the following example:

```
new CustomControl<bool>(polygonsCheckBox, s,
true).setCustomEventHandler(boolUpdateVis);
```

The *boolUpdateVis* procedure for this control is defined as follows:

```
CustomControl<bool>.CustomEventHandlerProc boolUpdateVis = delegate(object sender,
System.EventArgs e, bool defaultValue) { mCore.updateVis(); };
```

The following snippet shows the corresponding condensed version:

```
new CustomControl<bool>(polygonsCheckBox, s,
true).setCustomEventHandler(delegate(object sender, System.EventArgs e, bool
defaultValue) { mCore.updateVis(); });
```

Special Case *CheckedListBox*

The checked lines of a *CheckedListBox* control (wrapped by *CustomControl<string>*) are handled in the form of space-separated tokens in a string, for instance *"2 5 7"* if the items 2, 5 and 7 of the list 1, 2, 3, 4, 5, 6, 7, 8 are checked.

Example:

```
new CustomControl<string>(numSimSigListBox, s, "2 5 7");
```

Control Placement, *ResultsForm* Window

Preferably, custom controls are placed in the area below the *Visualization* box. For lack of space, additional controls may be placed in the *ResultsForm* window (button *Show Results+Nav Window*), which in any case shows log outputs generated using *System.Diagnostics.Debug.Print* statements, provided that the *DEBUG* constant is defined. If the *DEBUG* constant is not defined, the log output tab can be used for higher level information.

Example Implementation of a Custom Application

The symmetry detection outlined in the document *symmetry-detection.pdf* is implemented within the source code of this Visual Studio Solution and demonstrates the customization of *Dr. House*.

The following files are part of the example implementation and can be deleted together with the contents of custom procedures without affecting the common core application:

- *CustomCore.CurvatureTensorEstimation.cs*
- *CustomCore.HighlightBackgroundWorker.cs*
- *Diagram.cs*
- *GeoGen.cs*
- *KdTree.cs*
- *MeanShiftTools.cs*
- *Neighborhood.cs*
- *Symmetry*.cs*